Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-494-09277-7
Our file Notre référence
ISBN: 0-494-09277-7

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Dedication

Dedicated to my husband Dr. Mansoor Khan for all his love & support

ABSTRACT

Detection and correction of software faults is a difficult and complex issue. Mining software repositories is one of the approaches to assess and improve software quality.

The objective of our research was to mine software quality data from a large-scale open-source software system, Mozilla and its associated defect tracking database, Bugzilla.

The first contribution in this dissertation is the development of Entity Relationship diagram of Bugzilla database by performing data reverse engineering.

The second and most important contribution in this research is the development of Metrics-Delta data set, based on Mozilla source code and Bugzilla database. The data set contains 22 attributes and 8349 data points.

The final contribution is development of predictive models using support vector machine, neural networks and linear regression for Metrics-Delta data set. Predictive models can be used to predict software quality, which in turn minimizes failure rate and could be used as a management tool.

Acknowledgments

I would like to express my deep emotions to my parents Mr. Anwer Rajper and Dr. Mrs Rashida Anwer for all their love, care & support in every single moment of my life.

Great appreciation is due to my husband, Dr. Mansoor Alam, for all his support to make a dream come true and for always being there to count on.

Sincere thanks to my supervisor, Dr. Scott H. Dick, for his encouragement and valuable guidance while conducting this research.

Special thanks are due to my parents-in-law, Dr. Rahat Alam & Dr. Sabra Alam and my sisters Anita, Amber & Sana for their encouragement and appreciation.

And finally lots of thanks to my adorable kids Sunny Alam & Linta Alam for all their patience and understanding, during my research.

CONTENTS

CHAPTER 1

1. Introduction 1
2. Software Systems 5
   2.1. Object-Oriented Design 7
3. Software Quality 9
   3.1. Software Quality Attributes 10
4. Software Quality Assurance 11
   4.1. Software Testing 13
   4.2. Software Metrics 15

CHAPTER 2

1. Introduction 20
2. Knowledge Discovery in Databases 21
   2.1. Data Mining Techniques 24
      2.2.1. Neural Networks 25
      2.2.2. Support Vector Machines 28
         2.2.2.1. SMO Regression 30
3. Mining Software Repositories 33
   3.1. Defect Tracking 31
   3.2. Mining Defect Tracking Databases 35

CHAPTER 3

1. Introduction 38
2. Open Source Software 39
   2.1. Advantages of Open Source Software 43
   2.2. Disadvantages of Open Source Software 45
3. Mozilla- An Open Source Software 47

CHAPTER 4

1. Introduction 53
2. Data Reverse Engineering 53
3. Entity Relationship Diagram 55
4. Bugzilla Overview 56
5. Bugzilla Database 57
6. Entity Relationship Diagram of Bugzilla Database 59

CHAPTER 5

1. Introduction 67
2. Creation of Delta Data Set 70

2.1. Data Extraction 73
2.2. Data Transformation 74
3. Metrics Data Set 76
4. Metrics-Delta Data Set 79
4.1. Assignment of file-level defects to C++ classes 80
4.2. Joining Delta Data Set and Metrics Data Set 85

CHAPTER 6
1. Introduction 87
2. Data Mining 87
2.1. Variables and Description 89
2.2. Statistical Results 91
2.3. Experiments 96

CHAPTER 7
1. Summary 100
2. Future Directions 101

REFERENCES 104
APPENDIX A 114

LIST OF TABLES

Table 6.1: Statistical Results of Metrics-Delta Data Set 92
Table 6.2: Correlation of Metrics to Delta 93
Table 6.3: Pearson's Pair wise correlation 94
Table 6.4: Variable Abbreviations 95
Table 6.5: Regression Results 98

LIST OF FIGURES

Figure 1.1: Relationship of Test documents to testing process [53]	14
Figure 2.1:Overview of steps constituting the KDD process [34]	24
Figure 2.2: A 3-3-2 Neural Network [25]	27
Figure 2.3: A Linear Support Vector Machine [152]	30
Figure 2.4: Bug Life Cycle [150]	33
Figure 2.5: A test incident Report [138]	35
Figure 4.1: ER diagram of Bugzilla database pp.1	61
Figure 4.2: ER diagram of Bugzilla database pp.2	62
Figure 4.3: ER diagram of Bugzilla database pp.3	63
Figure 4.4: ER diagram of Bugzilla database pp.4	64
Figure 4.5: ER diagram of Bugzilla database pp.5	65
Figure 4.6: ER diagram of Bugzilla database pp.6	66
Figure 5.1: Sample Patch	72
Figure 5.2: Extracted Patched File Names	73
Figure5.3: Cleaned File Names	73
Figure 5.4: List of patched .h and .cpp Files	74
Figure 5.5: Mozilla Directory Sample	74
Figure 5.6: List of .h and .cpp files from Mozilla Directory	75
Figure 5.7: Delta Data Set	76
Figure 5.8: Joining Metrics-Delta Data Set	79
Figure 5.9: File Names, Class Names and SLOC-1	80
Figure 5.10: File Names, Class Names and SLOC-2	81
Figure 5.11: File Names, Class Names and TLOC	81
Figure 5.12: File Names, Class Names and Delta	82
Figure 5.13: File name, Class Name and Method Name-1	83
Figure 5.14: File name, Class Name and Method Name-2	84
Figure 5.15: File Name, Class Name and Contribution Factor	84
Figure 5.16: File Name, Class Name and Class-Level Delta	85
Figure 5.17: File Name, Class Name and Final Class-Level Delta	85
Figure 5.18: Metrics-Delta Data Set	86
Figure 6.1: Linear Regression Model	97
Figure 6.2: SMO Regression Model	98

CHAPTER 1

1. Introduction

Software systems have become an integral part of human life in 21st century. Their application starts from our kitchen to our lives. The diversity of application encompasses many areas, for example transaction systems, aircraft, real-time military operations, nuclear reactors, space programs, banking transactions, automotive mechanical and safety control to hospital patient monitoring systems and diagnosis services. The smooth functionality of these systems is essential. In order to get high quality performance from these systems; we need high quality software systems. In recent years the cost of developing these systems and the penalty cost of software failures has become the major expenses [35]. Some highlights of the consequences of software failures are:

- Failure of patriot missile in gulf war causing death of 28 American Soldiers [80].
- Ariane 501, a $500 million rocket self-destructs because of arithmetic overflow. The rocket exploded in less than 40 seconds after lift-off on 4th June 1996 due to design error and insufficient testing [106].
- Fifty thousands pieces of mail sent to the U.S Patent and Trademark Office was returned due to the deletion of zip code caused by software failure [23] of telephone systems for 10 million users.

These examples of software failures demonstrate the consequences caused by software failures. Failure of these systems may result in loss of life, damage or total destruction of a property, unachieved tasks, inconvenience and high cost maintenance.

Due to the increasing level of human dependence on software systems for almost every aspect of life, the problem of software quality is not just a luxury of research, but rather a necessity of the time. In all these cases the systems failure occurred due to the presence of faults in program source code [69]. In order to remove the faults, the basic requirement is their correct identification. Sooner the fault identification takes place; the better the chances are for higher level of quality product but there is an effort and cost of doing this. The definition of fault, its propagation and consequences were described in fault-error-failure model proposed by Vaos [38]. The model defines *fault* as a mistake introduced in the source code. This mistake could be a mistake in program coding, a logical flaw or a wrong interpretation of requirements.

Detection and correction of these faults is a difficult and complex issue. A study performed by Microsoft demonstrated the severity and complexity of the problem. The results of the study showed that it takes about 12 programming hours to locate and correct a software defect. At this rate, it can take more than 24,000 hours (11.4 man-years) to debug a program of 350,000 lines of code with a cost of US $ 1million [105]. Large software systems like space shuttle programs or antiballistic missile system contains at least 3 million and 10 million lines of code respectively [69]. The complexity and size of the systems are the root cause of the problem.

In order to improve software quality, identification of individual modules containing faults among thousands of source files is a difficult job. The amount of test required to test the software system or to maintain the existing system have been a challenge for project managers. Software maintenance consumes most of the resources in many organizations.

There are many approaches, which address to the problem of achieving high quality software systems and many yet to explore. One of the approaches is knowledge discovery in databases (KDD). KDD is the process of identification and extraction of useful information from large amounts of data and one of the benefits offered by applying KDD process is software quality prediction. Metric values collected from each code model could be used as a data set for the KDD process [102]. The two main questions related to the use of metrics for data mining process that offers point of interest are

- What metrics (features) are indicators of high quality system?
- What patterns of metrics indicate potentially high defect or high- risk modules?

Shin and Goel [120] performed KDD on a NASA database of software metrics. Khoshgoftar [60] has also proposed the conduction of KDD process for software quality prediction using software metrics. They conducted case study on large telecommunication system and collected software metrics from its source code, configuration management transactions and problem reporting transactions. Using CART to develop predictive model, they predicted which modules are likely to have faults.

The recent work [23,121,127,2,137,39,125,132,24] of performing KDD on software repositories is gaining interest. Software repositories hold valuable data and the information extracted from them can help in defect analysis, software process control, software reuse and system understanding. Defect tracking systems (one of the type of software repositories) are of special interest for mining. Defect tracking systems allows individual or group of users to keep track of the outstanding bugs (defects) in their product [8]. They not only allow recording of the software faults but also keep track of actions taken to repair them. Mining defect tracking databases can provide useful

information for assessing software quality, fault prediction and can also guide to develop new tools for bug determination [121].

The objective of our research was to mine software quality data from a large-scale open source software system Mozilla to predict software quality by conducting KDD process proposed by Fayyad [34]. The KDD process defined by Fayyad [34] is composed of nine sequential activities: learning the, application domain, creating a target data set, data cleaning and preprocessing, data reduction and projection, selection of data mining function, selection of data mining algorithm, data mining, and evaluation and use of discovered knowledge and we have followed them religiously. The KDD process is performed on software metrics derived from Mozilla source code and its associated defect tracking database-Bugzilla.

The first contribution in this dissertation is the development of Entity Relationship (ER) diagram for providing the conceptual model of Bugzilla database. As discussed earlier the KDD model proposed by Fayyad [34] is followed in this research and the first step specified in that model is to understand problem domain. The ER diagram was required since Bugzilla database belonged to an open source project and one of the problems associated with open source projects is lack of documentation. The complex database was hard to understand without any conceptual model. The process of data reverse engineering is conducted to develop the ER diagram. The reverse engineering of databases is opposite to forward engineering. While forward engineering starts from developing conceptual model of system according to requirement specifications and produces physical database as an end result, reverse engineering starts by examining physical database and produces conceptual model as an end product. We examined

Bugzilla source code and developed the ER diagram by careful observation of data definition statements. The details are provided in chapter 4.

The second and most important contribution in this research is the development of data set named Metrics-Delta data set based on Mozilla source code and its associated defect tracking Bugzilla database. The data set is created in three steps. First, using an instance of Bugzilla database identification and extraction of number of defects (Delta) for C++ source code and header files for is performed. Secondly, software metrics are collected from Mozilla source code using Krakatau for C++ classes and methods. And finally using file name as a key defects are mapped from files to individual C++ classes. Chapter 5 contains all the relevant details of Metrics-Delta data set.

The third and last contribution is developing a predictive model using linear regression, support vector machine and neural networks for Metrics-Delta data set. The algorithms applied are SMO regression [110] and multilayer perceptron [54] using the data mining tool WEKA. The experiments are conducted using ten-fold cross validation. Predictive models and results are given in chapter 6. The predictive model can be used to predict which modules in Mozilla are risky. Prediction of defect prone modules minimizes failure rate and could be used as a management tool.

In the remainder of this chapter an overview of software systems, software design and software quality assurance is given. Followed by the usefulness of software metrics and details of object-oriented metrics.

2. Software Systems

Software systems are different from hardware systems. Hardware systems design is guided and limited by natural laws of materials whereas software systems design do not

have any natural limits and are flexible and malleable in nature. While there are errors in many engineering products, experience has shown that errors are more common, more pervasive, and more troublesome, in software then in other technology [63].

Software systems due to their logical nature inherit four problems that are *complexity, conformity, changeability & invisibility* [14]. There are many contributors that make the software systems *complex* like large number of possible states, hard to use functions, program extensions and size of the software system and number of interfaces. *Conformity* refers to the constraints placed by complex human institutions and systems (e.g., the tax regulations of a state, pre-existing hardware, third party components and business rules) on the software systems. *Changeability* is often introduced in software systems due the need of additional features. Successful software systems could be subject to change to enhance its capabilities, or even apply it beyond the original domain, as well as to enable it to survive beyond the normal life of the machine it runs on and to be ported to other machines and environments. Where as the *invisibility* is because of difficulty in software visualization. It is hard to represent the conceptual picture of any part of the system developed in one person's mind, which often leads towards communication gap.

There is no geometric representation for complex software system as is available for any mechanical, electronic or construction systems.

The design of software systems is a continuing evolutionary discipline that is making software systems advance and complex day by day. The design of earlier software systems was focussed on developing modular programs in a top down fashion, which was then followed by procedural design. Procedural design evolved the concept of structured programming. Structured programming offered a disciplined approach of writing

programs that could be easily test, debug and modify. However, the incapability of structured approach to mirror real-world entities effectively raised the question that this programming approach is not effective to produce a reusable code and incase of modifications, the start from scratch problem arise [28]. Object-Oriented design provided the solution to many problems faced by its preceding design approaches. It has been evolved over past three decades and its unique nature lies in its ability to support three fundamental design concepts: abstraction, information hiding and modularity.

2.1 Object- Oriented Design

An Object-Oriented Design (OOD) is an approach to software development in which the structure of the software is based on *objects* interacting with each other to accomplish a task [21] where as the object is a component of real world mapped into the software domain. These objects are created from classes, which are a self-contained description for a set of related services and may be invoked by a message (a request to an object to perform one of its operations). As discussed earlier the uniqueness of OOD is due to its support for three basic design characteristics that are Encapsulation, Inheritance and Polymorphism.

Encapsulation means that each object hides as much from the outside world as possible, principally the data that describes it, and the internal processes that enable it to perform the functions for which it is designed. It provides higher level of abstraction for a class by encapsulating the object's internal details (both behavior and attributes). Since the implementation details of an object are hidden, the users of the object need to know only how to use the object rather than knowing how object performs its operation. For example, a file object will probably expose pieces of information via access methods

such as its name and size, and data can be passed to it, and retrieved from it using other methods, but a programmer wishing to use the object need not know where or how the data is stored . In object orientation, the application of the concept of encapsulation is not only restricted to the composition of classes and objects but also includes higher level of encapsulation like forming packaging and subsystems. It simplifies software development and increases the potential for code reuse.

Inheritance is the form of reusability in which programmers create classes that absorb existing class's data and behaviors and enhance them with new capabilities. It allows the class to build a hierarchy. The child class can inherit features from the parent class thereby reducing an extra effort to build the feature, which already exists. The sub-class can inherit both behavior and attributes from its super-class. Inheritance enables designers to create many objects from the templates of actual objects just like the architect can build many houses from the architectural map of one house identically or by little modifications for example adding extra features. For example in geometry, a rectangle is a quadrilateral. Thus, in C++, class *Rectangle* can be said to inherit from class *Quadrilateral*. In this context class *Quadrilateral* is a parent class and class *Rectangle* is a child class.

The property of an object to exhibit different behavior on different class or sub-class is called *Polymorphism.* In particular, polymorphism enables us to write programs that process objects of classes that are part of the same class hierarchy as if they are all objects of the hierarchy's base class. Polymorphic operations are mapped to the correct method by the examination of the signature calling the operator [30.] Two methods

similar in nature logically may fall under same name, but due to the difference in parameter value, they can create different objects.

The approach of software reuse was introduced by OOD and is now extended to component-based design (CBD). A component can be defined as "a reusable part of software, which is developed independently and can be combined with other components to build larger units" [105] and can be accessed via interface. The interface is clearly separated from a component. The most important feature of component is the separation of its interfaces from its implementation there by causing the component as a black box with explicit encapsulation boundary. Different component technologies currently available in the industry are JavaBeans, COM+ and CCM.

The support of CBD to join two or more components easily in order to build a larger component has lead towards the frameworks. A framework supports the software reuse by providing a skeleton of an application, which can be modified within a certain domain. This modification can be made during program design or execution and is referred as instantiation of framework. Examples of component-based framework are .NET and J2EE.

3. Software Quality

The meaning of word quality defined by Oxford English dictionary is "degree of excellence". This definition applies for defining quality of the physical systems but as far as quality of software systems is concerned, it seems to be insufficient. A formal and detailed definition of quality stated by International Standard Organization (ISO) is " The totality of features and characteristics of a product or service that bear on its ability to

satisfy specified or implied needs (ISO 1986)" [21]. This definition provides more details about the term quality but if we observe carefully this statement is pointing towards two features, which a quality product should posses. First the perfection of features (same as word meaning) and second, fulfillment of desired user satisfaction.

USA Department of Defense (DoD 1985) has defined software quality as the degree to which the attributes of the software enable it to perform its intended purpose. The major hindrance in achieving an ideal quality product is the complexity of the software systems, which makes software systems development and testing time consuming, expensive and error-prone task.

3.1 Software Quality Attributes

Software quality can be measured by number of quality factors. Mc Call, Richard and Walter [21] proposed the software quality model named as Mc Call's quality model, according to which software quality can be measured in terms of correctness, reliability, efficiency, usability, maintainability, flexibility, reusability and portability. Mc Call has defined these attributes as quality factors.

- Correctness-The extent to which program fulfills its specification.
- Reliability- A measure of the rate of failure of the system.
- Efficiency- the amount of computing required performing the function.
- Usability-Effort required learning, operating, preparing input & interpreting output of program.
- Maintainability- Effort required locating and fixing errors.
- Flexibility- Effort required modifying an operational program.
- Reusability- the extent to which the program is reusable.

- Portability- Effort required for migrating the program.

Although every quality attributes posses its own importance but reliability is directly related to safety and availability of the system. The lack of reliability can cause the system failure, which in turn can cause the hazardous accidents or make the system unavailable to its users. Availability of the system is the attribute that defines the readiness of system usage and in many cases posses utmost importance whereas safety is concerned with occurrences of mishaps. Quantifying every quality measure is a difficult job. Every software system is built for some specific purpose and so the quality priorities vary according to individual product specification. The particular quality measure (either one or more than one) is related to the satisfaction of specific task while doing tradeoff for some other quality measure. For example, it is possible that reliable software could not be flexible that is reliability is achieved on the cost of flexibility.

4. Software Quality Assurance

Software Quality Assurance (SQA) is the planned and systematic set of activities that ensure that software process and products conform to requirements, standards, and procedures. "Processes" include all activities involved in designing, coding, testing and maintaining; "products" include software, associated data, documentation, and all supporting and reporting paperwork [57]. A typical quality assurance process includes variety of tasks associated with seven major software quality assurance (SQA) activity [108].

- *Application of technical methods*: Quality is build into the system by applying quality approaches in all phases of software life cycle. The first step in quality assurance

activity is selection and use of technical methods and tools that helps analyst to measure the quality of the product.

- *Conduction of formal technical reviews*: Technical reviews help in identifying quality problems related to design and should be carried out by technical staff.
- *Software testing*: Software testing is a series of testing activities required identifying and removing the software faults. It is carried by developers and testers and is usually composed of unit testing, integration testing and system testing.
- *Enforcement of standards*: The formal standards and procedures of software engineering process vary from company to company. The SQA activity is required to measure the degree to which the standards are followed.
- *Control of change*: Changes are required in almost every software system. There could be number of reasons to that. The most common is due to the change of requirements or for the request of additional features. The impact of these changes needs to be monitored closely and the formal change control process is implemented. During the change control process, the formal change requests are analyzed and evaluated and the impact of change is controlled.
- *Measurement:* The measurement of the software is an integral part for software quality assurance. Metrics are collected for technical and management support and decisions.
- *Record keeping and reporting*: The results of performing reviews, audits, change control, measures and testing must be properly documented to be refereed when desired.

All of these activities are addressed to ensure the quality of the product according to quality standards. Along with the organizational internal standards, there are some international standards too like IEEE standard and ISO 9000 standard for software quality assurance. As discussed earlier in this section that software quality assurance is composed of series of activities including software testing and measurement. Software testing generates bug reports we use to count defects, while measurement provides software metrics. Sections 4.1 and 4.2 describe the details of software testing and software metrics respectively.

4.1. Software Testing

Software testing is a critical element of software quality assurance and presents an ultimate review of specification, design and coding [108]. It is the process of establishing confidence that a product satisfies its intended purpose or not [3]. During testing process with the help of manual or automated means the actual output is compared with the desired output. There are several techniques to assess the software quality. These include structured walkthroughs, software inspections, static analysis, dynamic testing, symbolic execution and proofs of correctness [36]. The selection of testing process varies from organization to organization but it remains the unavoidable in any system. We cannot actually predict how a program will behave without actually running it. If testing is conducted successfully it will detect errors in the software, checks the fulfillment of requirements and provides indication of software quality. The IEEE standard for software test documentation 829-1998 [53], provides a flow chart [see fig 1.1] for the specification of test documents and testing process. The following activities should be performed sequentially to implement test standard.

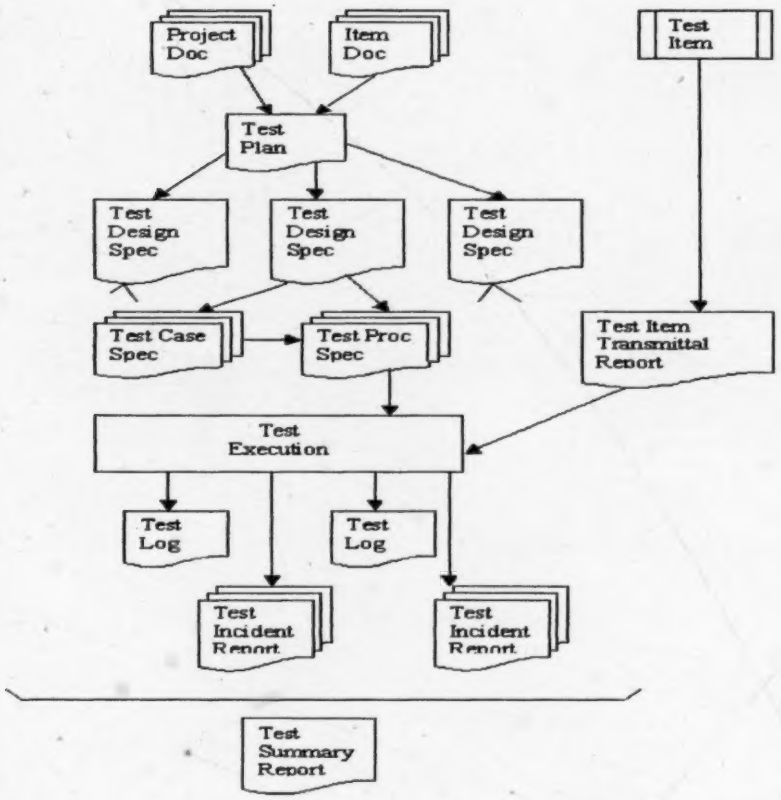

Figure 1.1: Relationship of test documents to testing process [53]

Testing activities should be started by developing a *test plan*. Test plan is a document, which defines the scope, approach, resources and schedule of the intended testing activities. It provides all the details of targeted test items including what features to be tested that will perform testing and outlines if there are any risk factors associated with the testing process. Followed by test plan, *Test design specification* should be conducted. Test describes the testing approach for a particular feature or features along with test identification. The third activity in this sequence is development of *test case*

specification, which describes details of inputs, expected results and list of execution conditions. *Test procedure specifications* specify sequence of actions to conduct a test. The next step is the development of *Test item transmittal report*. This document identifies the test items along with their current status and location information. The following document that is *test log* gives the details of all the records about test execution in chronological order. The last two document that are *test incident report* and *Test summary reports* are used to specify the occurrence of an event (bug) during testing process and the details of the occurred events respectively.

Once incident reports have been filled out and reported defects (bugs) have been resolved, the history of defect tracking system can help in identification of types of defects reported frequently and which modules are likely contain them.

4.2. Software Metrics

Software Metrics are used to identify modules in software systems that are potentially error prone, so that the extra development, testing and maintenance effort can be directed towards the faulty modules. Metrics are aspects of software development (either some part of the software product itself such as code or the documentation, or the development processes producing that product such as coding or testing phases) that can be measured. These measurements can be used as variables (both dependent and independent) in models for predicting or estimating some aspects of the development process or products that are of interest such as effort, errors remaining in the system after testing, user satisfaction, or system performance. Each software metric quantifies some characteristic of a program. Different types of metrics are available to describe different characteristics. Like number of lines in a source code, number of operands (Halstead's number of

operators) or number of comments. Although these things appear to be simple but they help to develop a picture of the program, like number of lines describes the length of the program [45]; number of operands describes the amount of computation lies in the program. Another important example is object-oriented metrics.

Object-oriented metrics are the measures of software objects. Shyam R. Chidamber and Chris F. Kemerer [20] proposed a first set of six object-oriented metrics namely DIT, NOC, WMC, CBO, RFC and LCOM. The Object-oriented metrics provides insight view of the object-oriented software. They can be used for measuring complexity, class semantics and relationship between classes. They can play useful role for software management like measuring software process, identifying design flaws and effort required for testing. Chidamber has defined object as a substantial individual and collection of all of its finite properties.

- DIT-depth of inheritance: Depth of a class in a tree is the length of maximal path from node to the root of the tree.
- NOC- number of children: Number of descendents of a class.
- WMC-weighted method per class: WMC is the sum of complexities of a method. If a class has M_1, ..., M_n number of methods and complexities of the methods are given by $C_1, \ldots, C_n$, then :

$$\text{WMC} = \sum_{i=1}^{n} c_i$$

If all complexities are considered to be unity, then WMC = n, number of methods.

- CBO- coupling between objects: Two objects are coupled, if at least one of them is dependent on other.

- RFC-response for class: Set of all methods that can be invoked by the object of a class.

- LCOM-lack of cohesion in methods: LCOM is a count of the number of method pairs whose similarity is zero. It is obtained by subtracting number of similarities between methods from number of non-similarities between methods. *Given n* methods M_1, $M_2, \ldots, M_n$ contained in a class C_1, which also contains a set of instance variables $\{I_i\}$. Then for any method M_i we can define the partitioned set of

$$P = \{(I_i, I_j) \mid I_i \cap I_j = 0 \text{ and } Q = \{(I_i, I_j) \mid I_i \cap I_j \neq 0$$

then, LCOM = $|P| - |Q|$, if $|P| > |Q|$

=0 otherwise .

The value of DIT provides the picture of class complexity thereby suggesting the amount of testing required for that class. In their case study [20], they observed the presence of a class in their test data, which consisted only 4 methods but objects of that class inherited 132 methods from its hierarchy. The testing of that class is complicated rather than its design. Another measure of inheritance is provided by NOC. A class with higher number of children provides more chances of reuse but on the other hand reduces the abstraction of base class. Thus providing an option to software designers to tradeoff according to their own requirement. WMC measures complexity of a class. A class with large number of methods might have no children and on the other hand a class with less number of methods might have many children. In that case testing of a class with higher number of methods will require less testing effort as compared to class with less number of methods

but higher children. CBO provides the measure of coupling. A class with higher value of CBO might violate the concept of encapsulation and therefore should be minimized in order to prevent integrity. RFC captures the level of communication between classes by measuring the inter-class couples and methods external to the class. Its value is useful in determining test resource allocation. The last metric described in that suite is LCOM. LCOM is a measure of cohesiveness. The lower value of LCOM supports class's encapsulation and its higher value is the indication that a class is attempting to achieve many goals and there is a chance error and requires careful testing.

The CK metrics suite provides a measure to check the integrity of any object-oriented design. From management and designers point of view they are useful to measure architectural and structural consistency of the application. The metrics can be used to identify outlying classes for special attention by putting some threshold values for comparison.

Software systems are complex artifacts and their complexity is increasing with the innovation of technology. In order to make these systems dependable software quality assurance is mandatory. Software quality assurance can be achieved by performing set of activities and according to the scope of this dissertation software testing and software metrics are described in this chapter. Since we have collected object-oriented metrics from Mozilla source code, a description of classical CK metrics is also included in this chapter

The objective conducting of this research is to mine software quality data from a large-scale open source software system, which was achieved by following KDD model proposed by Fayyad [34]. The contents of chapter 2 are therefore devoted to the

description of Fayyad's KDD model along with the details of mining software repositories and mining defect tracking systems. Chapter 3 is focused open source software development. The advantages and disadvantages of open source software are discussed along with detailed overview of Mozilla. Chapter 4 contains details of data reverse engineering, overview of Bugzilla database and ER diagram of Bugzilla database. Chapter 5 is devoted to the development details of data set. Finally, in chapter 6 the results and details of predictive experiments are provided while chapter 7 is dedicated to concluding summary and future directions.

CHAPTER 2

1. Introduction

Across the world, in almost every dimension of life, data is collected and stored either manually or electronically. Electronic storage normally implies database systems, which store these facts about the real world in structured and organized manner however, the size and complexity of database depends on the volume and nature of data. Current hardware and database technology allows efficient and inexpensive reliable data storage and access [34]. For example if we consider the example of United States Internal revenue agency and assume that there would be 100 million tax payers and each tax payer fills five form with approximately 200 characters per form, we could get a database of $100*10^6*200*5$ characters of information. If we assume that the IRS keeps the past three returns along with the current one, then we are looking at a database of 400 gigabytes [32]. Scientific instruments can easily generate terabytes and petabytes of data at the rates as high as gigabytes per hour [35]. Due to this increased potential of collecting and generating data, the size of databases has increased dramatically. They are increasing in size in two ways: (1) the number N of objects or records and (2) the number d of attributes [34]. Other examples of large databases are super markets store electronic copies of million of receipts, bank and credit card companies maintain extensive transaction histories government organizations store data about millions of citizens and medical diagnostic applications.

The large databases or information repositories contain patterns and regularities in data, which if explored provide rich and useful information. No matter what is the source of

data whether it has come from science, business or government the data themselves (in the raw form) are of little direct value. What is of value is the knowledge that can be inferred from the data and put to use. The identification of knowledge to be extracted and the transformation of data into desired information are the crux of conducting performing any data mining project.

The extracted knowledge can be applied for vast variety of applications like information management, decision-making, process control [77] and quality prediction. The traditional method of turning data into knowledge relies on manual analysis and interpretation, which is slow, expensive and highly subjective [124] particularly if we are looking at large databases, the manual extraction becomes quite impractical. There is a gap between data collection capabilities and data analyzing capabilities [34]. The volume and dimensionality of data are the root problem [35]. The need to scale up the process with the help of automated or partially automated techniques lead towards KDD.

2. Knowledge Discovery in Databases (KDD)

The phrase knowledge discovery in databases was coined at the first KDD workshop in 1989 to emphasize that knowledge is the end product of data driven discovery [124]. A more formal definition of KDD is "Knowledge discovery in databases (KDD) is the process of identifying valid, novel and potentially useful information from data in the context of large databases " [120]. In other words the goal of knowledge discovery is to find interesting patterns or models that exist in databases but hidden among volumes of data with the help of particular data mining methods. The KDD process accomplishes this task by using data mining algorithms to extract the hidden knowledge, using a database along with pre-processing, sampling and transformation of that database. The basic

problem addressed by the KDD process is one of mapping low-level data into other forms that might be more compact, more abstract and more useful [124].

The terms KDD and data mining are sometimes used interchangeably but data mining is one step in the KDD process for recognition of patterns or modules [34]. KDD in general and data mining in particular, is focused on finding patterns and models that can be interrupted as useful knowledge. No matter what the technique is used, the unifying goal is extraction of useful knowledge.

The KDD process is comprised of the following steps [34] (see Figure 2.1).

1. Learning the application domain: understanding application domain, relevant prior knowledge and objectives of end –user.
2. Creating a target data set: selecting the variables and record on which knowledge discovery is to be performed.
3. Data cleaning and preprocessing: Identification and removal of noise, consistency checking, strategies for handling missing data fields and preprocessing.
4. Data reduction and projection: Finding useful feature to represent the data, eliminating the irrelevant variables and appropriate transformations.
5. Choosing data mining function: making decision about the process goal that is description or prediction. The major classes of data mining methods are predictive modeling such as classification and regression; segmentation (clustering); dependency modeling such as graphical models or density estimation; summarization such as finding the relations between fields, associations, visualization; and change and deviation detection/modeling in data and knowledge.

6. Choosing data mining algorithm: selecting methods to be used for searching patterns in data
7. Data mining: Searching of patterns in a particular representational form or in a set of such representations, including classification rules or trees, regression, clustering, sequence modeling, dependency and line analysis.
8. Interpretation or use of discovered knowledge: Interpreting mined patterns and consolidated discovered knowledge.
9. Using discovered knowledge: Incorporating or using the extracted knowledge into the system for which KDD process is carried out.

The data mining process can be accomplished by number of techniques like machine learning, pattern recognition in databases, statistics, artificial intelligence, and knowledge acquisition for expert systems and data visualization [77]. Data mining is thus a multidisciplinary approach representing a catchall for a wide variety of techniques, working together for a unifying goal of discovering useful and valuable information, in the form of patterns and rules, from relationships between data elements. It's accomplished by a discovery-driven approach, whereby no a priori hypothesis is stated for a particular problem under investigation [59].

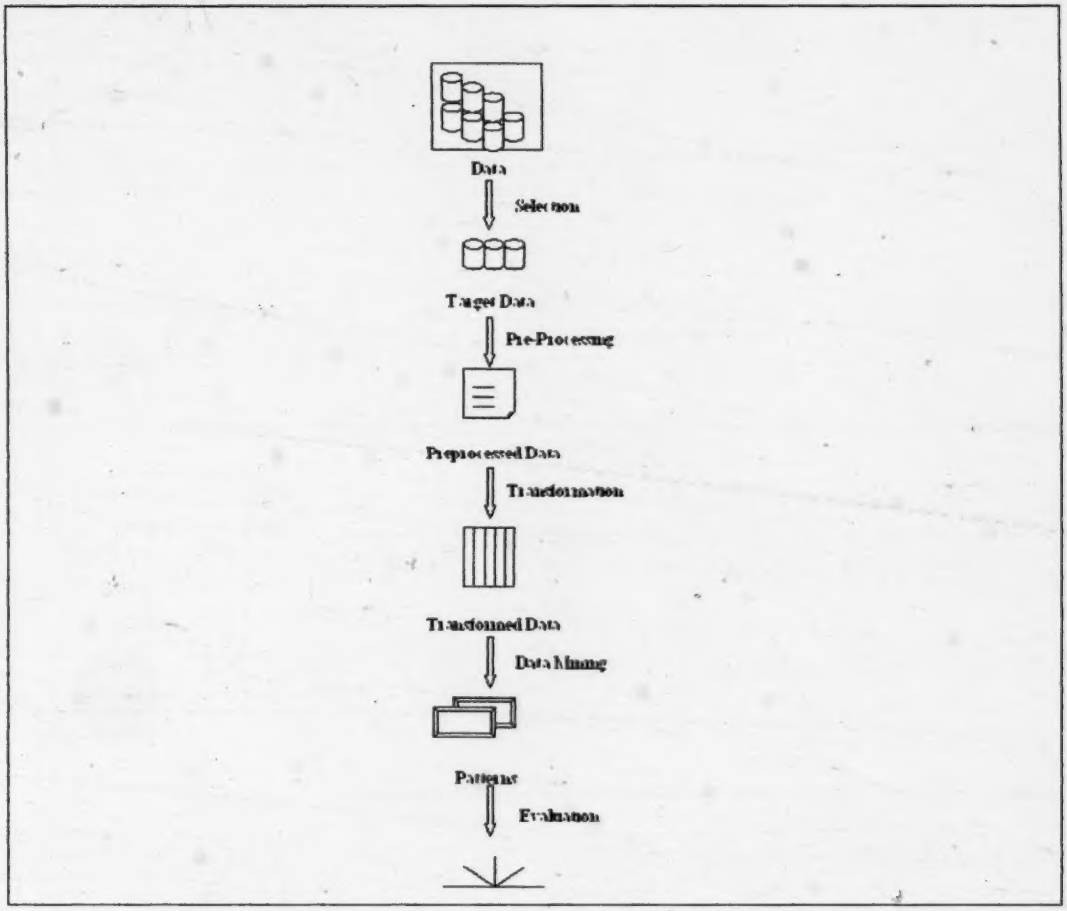

Figure 2.1: Overview Of Steps Constituting The KDD Process [34]

2.1 Data Mining Techniques

Data mining step of the KDD process refers to the process of applying discovery algorithm to the data. The data mining process can be broadly categorized into two main categories: predictive data mining and descriptive data mining. *Prediction* involves using some variables or fields in the data set to predict unknown or future values of other variables of interest. *Description*, on the other hand, focuses on finding patterns describing the data that can be interpreted by humans. The goals of prediction and

description are achieved by using appropriate data-mining techniques. The most common data mining goals are are [76, 34].

- *Classification* – discovery of a predictive learning function that classifies a data item into one of several predefined classes.
- *Regression* – discovery of a predictive learning function, which maps a data item to a real-value prediction variable.
- *Clustering* – a common descriptive task in which one seeks to identify a finite set of categories or clusters to describe the data.
- *Summarization* – an additional descriptive task that involves methods for finding a compact description for a set (or subset) of data.
- *Dependency Modeling* – a description for finding a local model that describes significant dependencies between variables or between the values of a feature in a data set or in a part of a data set.

Data mining can be performed by number of methodologies and techniques. The consecutive section contains description of neural networks and support vector machines because they are used in data mining step of the KDD process in this research.

2.2.1 Neural Networks

The idea of neural networks was inspired by modeling human brain at low level. The objective for establishing an artificial neural network is learning to learn the real world model (environment) in which it is embedded and to improve its performance through the learning process [76].

It is a set of interconnected nodes each having a number of inputs, an output and a transformation function. Each node possesses a processing unit, and the links between

nodes specify their relationship. These nodes are adaptive in nature, which means that the output of these nodes is dependent on modifiable input parameters. The adjustment of parameters is determined by the learning rule (a mathematical expression), in order to minimize the error rate.

The elementary component of every neural network is called a neuron. A neuron is an information-processing unit that is composed of a set of *input*, an *adder* (for summing input weight signals) and a nonlinear *activation function* that determines the output of a neuron [76].

Rosenblatt [54] first applied single-layer perceptrons to pattern classification learning in the late 1950s. During early development stage they were restricted to single layered systems however they failed to gain any successful results because of their limited learning capabilities [76]. Later on the research lead to the evaluation of multi-layered systems followed by the use of back propagation learning algorithm in multi-layer perceptrons by Rumelhart (1983) [54]. Multiplayer perceptrons using Rumelharts back propagation architecture are accepted as very powerful and flexible inductive learning algorithms.

The architecture of a multilayer perceptrons can be described as one layer for the input, one for output variables, and between these layers at least one hidden layer (see Figure 2.2). Each neuron in the input layer receives one input variable. The output of the input layer later becomes input of the first hidden layer. Each and every input in the hidden layers and output layers has an associated connection weight. The pattern of the connection weights is the neural network model of the real world problem. Finally the output layer neurons provide the output variable values. A network learns by finding a

vector of interconnected weights that minimizes its error on the training data set, by predicting the output for a given input, and comparing this prediction to the known correct value.

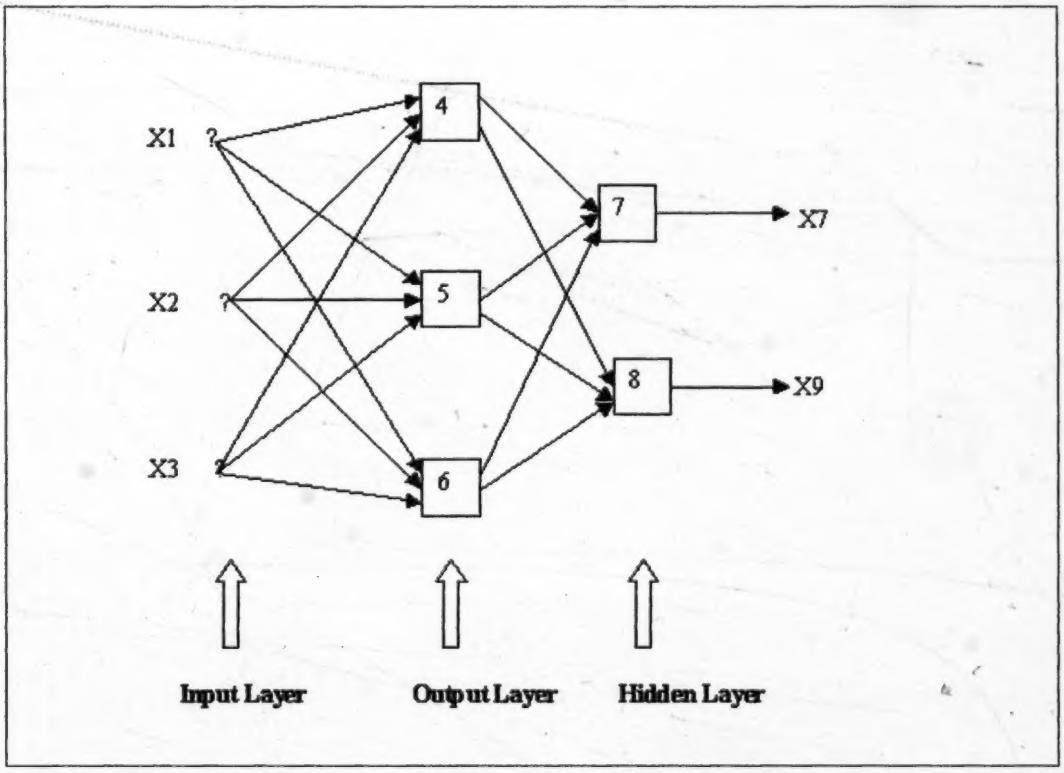

Figure 2.2: A 3-3-2 Neural Network [54]

Due to adaptive nature of neural networks they are good at pattern recognition. Neural networks have been used in software metric modeling [78]. Gray and Mac Donell [43] described guidelines for model development of software metrics using various neural network architectures, and Li [72] provides a good general introduction to neural network applications. Neural networks have been used successfully in many software metric-modeling studies, including Witing [131], where prediction accuracy was within 10%.

2.2.2 Support Vector Machines

The support vector machine (SVM) was first invented by Vladimir Vapnik in 1979 [110]. It enables the learning machine to generalize unseen data by dividing a hyperplane into positive and negative values. They are widely used these days for handwritten character recognition, text categorization, image recognition and bioinformatics [18].

The linear form of SVM performs (see Figure 2.3) the generalization with maximum margin where margin is the distance of hyperplane to the nearest positive or negative point and is given by

$$u = w \,.\, x - b, \tag{1}$$

Where w is the normal vector and x is the input vector in the hyperplane. The separating hyperplane is the plane where u=0 and thus the nearest points are given by u=1 and u= -1. Therefore the margin m is thus given by

$$m = \frac{1}{\|w\|_2} \tag{2}$$

The maximum margin can be expressed by the following optimization problem [117],

$$\min_{w,b} \frac{1}{2}\left\|\vec{w}\right\|^2 \text{ subject to } y_i(w.x_i - b) \geq 1, \forall_i, \tag{3}$$

Where x_i is the i^{th} training example and y_i is the i^{th} training output. This optimization problem can be converted into dual form, which is a quadratic programming (QP) via Lagrangian multipliers ($\propto_i$).

$$\min_{\alpha} \psi(\alpha) = \min_{\alpha} \frac{1}{2}\sum_{i=1}^{N} \sum_{j=1}^{N} y_i y_j (x_i \cdot x_j)\alpha_i \alpha_j - \sum_{i=1}^{N} \alpha_i, \tag{4}$$

Where N is the number of training examples.

$$\alpha_i \geq 0, \forall_i, \tag{5}$$

$$\sum_{i=1}^{N} y_i \alpha_i = 0. \tag{6}$$

There is one to one mapping between Lagrangian multipliers and training examples. Once lagrangian multipliers are determined, the normal vector w and the threshold b can be derived from them.

$$w = \sum_{i=1}^{N} y_i \alpha_i x_i,\ b = w.x_k - y_k \tag{7}$$

for $\alpha_k > 0.$

w can now be computed by equation (7). Equation (7) provides solution for linearly separable data however not all the data is linearly separable. This equation will lead to infinite solution for non-separable data. Cortes and Vapnik [126] proposed the modification to solution equation [3] by introducing slack variables. Slack variable allows margin failure and is given by

$$\min_{w,b,\xi} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N} \xi_i \quad \text{subject to } y_i(w.x_i - b) \geq 1 - \xi_i, \forall_i \tag{8}$$

Where C is the parameter which tradeoff wide margin with small number of margin failures. When the optimization problem is transformed into dual problem, the constraints in (5) changes to

$$0 \leq \alpha_i \leq C, \forall_i \tag{9}$$

the slack variable ξ_i do not appear in dual formulation. The output of non-linear SVM can be computed by Lagrange multipliers as

$$u = \sum_{j=1}^{N} y_j \alpha_j K(x_j, x) - b, \tag{10}$$

Where K is the kernel function that computes distance between input vector x and training vector x_j. The Lagrange $\propto_i$ are still computed by quadratic form. Although the non-linearities alter the quadratic form but the dual function remains quadratic.

$$\min_{\alpha} \psi(\alpha) = \min_{\alpha} \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} K(x_i, x_j)\alpha_i \alpha_j - \sum_{i=1}^{N} \alpha_i, \tag{11}$$

$$0 \le \alpha_i \le C, \forall_i \ , \ \sum_{i=1}^{N} y_i \alpha_i = 0.$$

The QP problem in equation (11) is solved by sequential minimal optimization algorithm (SMO).

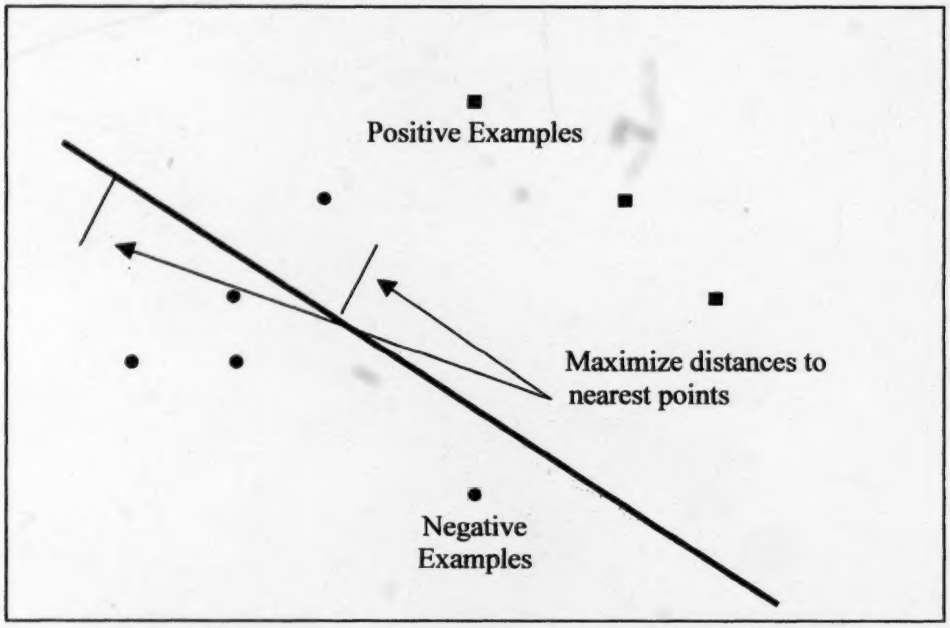

Figure 2.3: A Linear Support Vector Machine [126]

2.2.2.1 SMO Regression

SMO was first proposed by John C. Platt [110]. It is a training algorithm, which solves the quadratic programming problem of support vector machines by decomposing the

overall quadratic problem into small sub-problems. The SMO uses smallest possible quadratic programming problems that helps in obtaining quick and simple analytical solution. The SMO chooses to solve the smallest possible optimization problem at every step, which involves two Lagrange multipliers. The Lagrange multipliers must obey the linear equality constraint. At every step, SMO optimize two multipliers, find the optimal value and update the SVM to reflect the new optimal value. The key point is that for working set of 2 the optimization sub-problem can be solved analytically without explicitly invoking a quadratic optimizer.

The memory required by SMO is linear in size and due to this linearity it allows computation of large data sets.

3. Mining Software Repositories

The interest in mining software repositories for large long-lived projects is gaining interest. Software repositories hold valuable data, which if transformed into information can help in *defect analysis*, software process control, software reuse and system understanding. Current approaches on mining software repositories consist of ad-hoc scripts tailored to a particular data source. Those scripts manipulate the data source in the file system and produce metrics [2]. Examples of software repositories are Concurrent version control systems, Defect tracking systems and archived communications. Version control and bug tracking systems contain large amounts of historical information that can give deep insight into the evolution of software project [39]. Version control systems such as CVS [122], ClearCase [111] and Sourcesafe [86] hold information about evolutionary changes of a software project and are valuable source for retrospective analysis techniques which can explore for example change rates (number of changes

within certain amount of time and error proneness (number of errors) . Source code management systems, along with error tracking and change repositories maintain a comprehensive system history. They possess wealth of information regarding interactions and relationships of components. Data mining methods convert data containing past experience with a given process into knowledge about this process. Therefore source code management systems are a fertile area of application of data mining [121].

CVS, Concurrent Version Control Systems, is arguably the most widely used version control management system available in the market [23]. It distinguishes between version numbers of files (revision numbers) and software products (release numbers). For each working file in the repository CVS generates version control data and stores into log files [39]. CVS logs are a rich source of software trails. Software trails are defined as information left behind by the contributors to the development process such as mailing lists, web sites, version control log, software releases, and documentation and source code [23]. The information available in the logs can be very valuable for its developers, management and researchers to provide a fine-grained view of software project evolution. Frequency analysis of log messages identify the purpose of change, change size and time between changes [127]. Data extraction from CVS is very well covered and many tools are available for free for example SoftChange [122] is a tool that extracts and summarizes information from CVS and bug tracking systems]. Since we have performed a KDD process on defect tracking database the next section is designated to provide some introduction to defect tracking system followed by overview of research conducting in this domain.

3.1. Defect Tracking

Defect tracking is the activity of recording and tracking defects from the time they are detected until the time they are resolved [84]. Collection of defect information helps the project team to create graphs and reports that are useful for tracking and assessing project status as well as for developing a base of information that is useful on future projects.

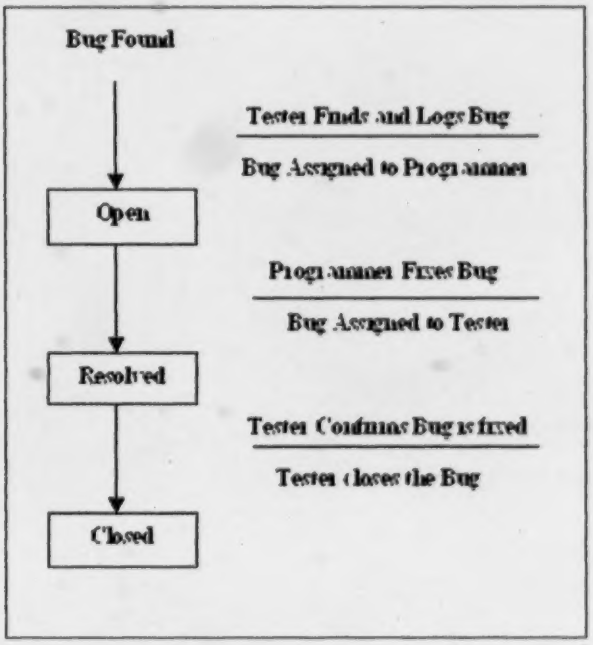

Figure 2.4 –A Bug Life Cycle [99]

The life cycle of a bug (see Figure 2.4) starts from its identification by the tester. After its identification it is defined as open and becomes available to the programmers to fix it. As soon as the programmer fixes the bug, it is passed to the tester for the confirmation of its fixing. Finally the tester has the authority to close the bug [116].

When a defect is found, an incident report should be filled out. The IEEE template for Test Incident Report 829-1998 [53] provides standards for the requirements of the incident report. An incident report should posses the following structure

a) Test incident report identifier:

b) Summary

c) Incident description

d) Impact

A generic term for incident report is modification request (MR). Each MR should posses a unique identifier, which can be a number or name. The incident summary should posses all the details about test case including test procedure specification, test case specification and test log [84]. The information in summary is commonly used for bug searches and it is important to be complete and concise [116]. The incident description describes details about inputs, expected results and actual results associated with the incident. It must contain the details of date and time of incident along with the details of testing environment and names of testers and observers as well (see Figure 2.5). Based on incident report and current development situation, the incidents are graded. The incidents are then fixed according to their relevant grades. Those incidents whose grading is high need to be fixed immediately while lower graded incidents are fixed with less priority [84]. The Impact of incident on test plans, test design specifications and test procedure specification is required to describe.

Defect ID (a number or other unique identifier)
Defect description
Steps taken to produce the defect
Platform information (CPU type, memory, disk space, video card, and so on)
Defect's current status (open or closed)
Person who detected the defect
Date the defect was detected
Severity (based on a numeric scale such as 1-4, or a verbal scale such as cosmetic, serious, critical, and so on)
Phase in which the defect was created (requirements, architecture, design, construction, defective test case, and so on)
Phase in which the defect was detected (requirements, architecture, design, construction, and so on)
Date the defect was corrected
Person who corrected the defect
Effort (in staff hours) required to correct the defect
Work product or products corrected (requirements statement, design diagram, code module, User Manual/Requirements Specification, test case, and so on)
Resolution (pending engineering fix, pending engineering review, pending quality assurance verification, corrected, determined not to be a defect, unable to reproduce, and so on)
Other notes

Figure 2.5: A Test Incident Report [84]

3.2. Mining Defect Tracking Databases.

Defect tracking systems allows individual or groups of developers to keep track of outstanding bugs (defects) in their product. They allow the development group not only to record the software faults but also keeps track of actions taken to repair them.

Mining Defect tracking databases can provide information, which could be used to assess software quality, fault predication and also can guide to develop new tools for bug determination.

In [96] the authors have developed a statistical model that uses fault information and file characteristics to predict which files of the system are likely to contain large numbers of faults. Testers can use this information to prioritize their testing and focus their efforts to make the testing process more efficient and the resulting software is more dependable. A tool is described and proposed in [125] to automate the prediction process. Mining past bug history of a software project can be used as a guide in determining what types of bugs should be expected in current snapshot and also can help in recommending which of a group of bug reports are more likely to be true [132]. A review on Apache web server [132] describes that bugs found in bugs database and by inspecting source code change histories are different in types and abstraction. Bugs found in bugs database were reported by outsider viewers and are reported against public release of the software rather than a CVS snapshot. These bugs are higher in nature because they usually point towards algorithmic flaws rather than simple coding problem. Based on types of bugs that are commonly reported and fixed in the code, we can determine what type of bug finding tools should be developed.

Another attempt on mining defect tracking database in conducted in [39]. The authors have proposed mining defect tracking database along with version control and have suggested that parsing the informal information contained in the modification report and linking them with data from bug tracking system could be useful for detection of

logically coupled files, identification of error prone classes with affected components or products, estimation of code maturity with respect to probability of remaining bugs and discovery rate of bugs in earlier releases of the system [39]. Fault history also provides valuable data for analysis of project trends and could be used to evaluate new projects [125].

This chapter was focused to the overview of KDD process along with the details of mining software repositories for extraction of software quality data. Mining software repositories specially mining defect-tracking systems can be useful for analyzing quality of the exiting system as well as for prediction of software quality. The next chapter is devoted to the discussion of open source software systems. Since we have used source code of an OSS system and its associated defect-tracking database to develop the data set, the details of the chapter 3 will help in understanding the background of OSS software, their associated advantages and disadvantages. Chapter 3 also includes the details of Mozilla, its provided tools and some description of technologies used in Mozilla.

CHAPTER 3

1. Introduction

Mining software repositories information can help in defect analysis, software quality prediction, software processes control, software reuse and system understanding. Examples of software repositories are concurrent version control systems, defect tracking systems and archived communications. Along with mining software repositories of conventional closed-source software, conducting KDD on open source software (OSS) system repositories has also gained interest.

The use of open source software (OSS) is growing. The open source software is available to download free of charge and its components may be integrated into other products. The software repositories of OSS systems can provide rich information due to number of reasons. Firstly, since OSS systems are released frequently, their CVS systems contain good amount of historical data. Secondly, because they are downloaded freely, the successful OSS systems have large number of potential users as well, which also take part in defect reporting, thereby making the defect reporting systems databases contain large amount of data. And finally, as the success of any OSS project depends on strong community formation, their archived communications are also good candidates for conducting KDD.

This chapter is focussed on providing the detailed overview of open source software, some popular open source products along with the discussion of advantages and disadvantages associated with it. Following by the details of OSS, the next section introduces Mozilla- an open source product. The introduction to OSS systems is given because we have used

Mozilla source code and its associated defect tracking system to construct the data set in order to conduct this research. Mozilla is a long-lived OSS system (more than eight years of age at the time of writing this thesis) and also a very large software systems. The Mozilla overview will provide the details of Mozilla including its development history, tools and architecture.

2. Open Source Software

The open source development has gained enormous attention over past decade and as an alternative to closed-source development is of particular interest to the software industry today [103]. The most well known attempt to informally define an open source process is Eric Raymond's '*the Cathedral and Bazaar*' paper [112]. The key ideas behind open source development process are [112]:

- The core system is developed locally by a single or team of programmers
- A prototype system is released on Internet, which others can freely read, modify and redistribute.

Open source developments typically have a central person or body that selects some subset of the developed code for the official releases and makes it widely available for the distribution. This is in contrast with traditional software development in several ways. The OSS development process is conducted by geographically distributed developers, in which work is not assigned to anybody but people undertake the assignments voluntary. There is no system-level designed or detailed design and no project plan, schedule, or list of deliverables [87]. Despite the lack of traditional approaches, open source software development is staggering. 30,000 projects are registered on http://freshmeat.net, about 70,000 projects are hosted on http://sourceforge.net and 5400 Perl modules are on

Comprehensive Perl Archive Network (www.cpan.com) [123]. Some popular open source projects are Linux, Perl, Apache, CVS, MySQL and Mozilla. Other open source projects are Topologilinux, Project @ssistant, Frozen Bubble, Tux Typing and Junit [41]. A case study and review on OSS projects [66] has classified the OSS (Open Source Software) into three main categories: Exploration-Oriented, Utility-Oriented, and Service-Oriented.

Exploration-oriented F/OSS projects, represented by GNU software and the Jun library, aim at pushing the frontier of software development collectively through the sharing of innovations embedded in freely shared OSS systems [66]. Project leaders, who are experienced programmers, are mainly responsible to develop and maintain the code. Active developers can participate in coding if their ideas are consistent with the project leader while the other community members acts as testers to provide the quality feedback. *Utility-oriented* OSS projects, represented by the Linux system (excluding the Linux kernel, which started as an exploration-oriented one and now is a service- oriented one), aim at filling a void in functionality. The project grows in "bazaar" style, in which peripheral developers modify the code or evolve new products from existing ones according to their own requirements. Little centralized control exists and project expansion is based on diversified needs and size of community. *Service-oriented* OSS projects, represented by PostgreSQL and Apache, aim at providing stable and robust services to all the stakeholders of OSS systems [66]. The population of stakeholders is larger than community that's why the version change is monitored very conservatively without affecting the core requirements of the stakeholders. Projects are governed by a

council of members rather than a single project leader. The members of the council are core developers who are senior developers and contributors to the OSS community.

An open source project focuses on growing its self-organizing community aspect, in order to get a significant success. The open source definition " work produced by a community of developers " [112] highlights that open source development is based on collaborative effort of group of developers. Usually it starts with a group of users, which later become part of the development and debugging process. The significant community is evolved using appropriate social and technical tools. Technical tools include mailing lists, newsgroups and source code management systems. Examples of social tools are differentiated roles and learning support.

One of the studies on understanding the motivations of people for involving themselves in the OSS projects has also revealed that the open source projects have a strong sense of community formation and adherence to norms of behavior [67]. The active open source projects have a well-defined community that's involved either in the development of the product or using the product [41]. In an empirical examination of open source project [65], the author has found that most of the open source products do not generate a lot of discussion and those projects who could generate a group or community are viewed and are downloaded more frequently. One of the important factors to be considered while selecting an open source product is it must have strong OSS community [115]. The case study on development process of Apache and Mozilla [87] pointed out that one of the key success factors for both of these projects is development of large and well-balanced communities. Mozilla also considers the effect of user participation as one of the key factors for its success [83].

The OSS community can be classified into two main categories [134] that are *user group* and *developer group*. The user group is further classified into Active users and Passive users.

- Active users are bug reporters and usually suggest new features and participate actively in forums and mailing lists whereas
- Passive users just download and use the products and are part of the user database.

The developer group can be categorized into four categories: Peripheral developer, Central developer, core developer and project leader.

- The Peripheral developers occasionally contribute in fixing bugs or in adding new features.
- Central developers actively contribute in fixing bugs, adding patches, writing supportive documentation and sharing and exchanging information.
- Core developers work as communicators between peripheral developers and central developers. They are council members, who extensively take part in the projects and manage the CVS releases.
- Project leader is usually the initiator of the project and in charge of decision- making. He accesses the feedback and directs the project with his own vision.

The OSS communities follow a life cycle and can be broadly specified into four main stages [68] namely Introduction stage, Growth stage, Maturity stage and Decline or Revival stage.

Introduction stage: The project is initiated by core developer or developers by launching

a working version of the software on the internet. One-on-one communication based on trust is conducted at this stage and the emphasis is on conveying the initial idea behind the project in order to develop the community. The organizational structure is informal at this stage.

Growth stage: As the project community increases, the needs for organizational structure usually moves initial developers to be the part of the management team along with their primary jobs. Project governance tools attract the bug reporters and fixers to be the part of the community and get identified. Communication forums provide platform for technical support, knowledge sharing and provide feeling of community.

Maturity stage: The number of members and downloads reaches to its maximum during this stage. The initiators become managers and are not involved in the development any more. They perform coordination within the organization, control the development process and perform communications at corporate level. The entire product is divided into modules and cooperating programs.

Decline or Revival stage: After reaching to the maturity stage a project might go to the decline stage. During decline stage the number of downloads and the number of community members decreases. In some cases there is a possibility of revival with the use of appropriate implementation and adjustment of governance tools.

2.1 Advantages of Open Source Software

The open source community claims a number of advantages over traditional software development. The first and most popular one (now a cliché in the open source community) is Eric S. Raymond's [112] "given enough eye balls, all bugs are shallow" approach. Since the product is freely available for use and observation, defects are found

and fixed quickly. Since this evolutionary process is based on the joint effort of a very large number of programmers rather than a limited development team, the resultant product is of better quality as compared to closed source products.

One of the open source projects is SAP, which is the primary downlink analysis and up-link planning tool for NASA's Mars Exploration Rover (MER) Mission. MER is a mission critical application whose failure can cause the crash of entire operation. The software was developed by using open source tools that are GNU Emacs editor, Concurrent Version System (CVS), JUnit, JavaCC and Xalan-J and OSS components that are Castor, Java Expression Parser (JEP), MySQL, MySQL Connector, HSQL Database Engine, Virtual Reality Modeling Language and Skaringa. The development team's comments about the quality of components they used are " The quality of open source components we used was excellent. In fact, overall they were of better quality than two commercial components we purchased for thousands of dollars". In their experience commercial companies take time to fix a bug, while open source developers do it immediately and enthusiastically. In one case, they diagnosed the problem, fixed it, and released a corrected version in less than a day [56].

The quick fixing issue puts the light on another open source issue, the motivation of the developers. Most open source developers are not paid and their goal of participation is personal satisfaction. A detailed study about motivation factors of open source software developers is reported in [67]. The motivation factors are categorized in two categories named as intrinsic motivation (the activity is valued for its own sake) and extrinsic motivation (providing indirect rewards for doing the task at hand). The study has concluded that the OSS contributors are motivated by a combination of intrinsic and

extrinsic factors with a personal sense of creativity being an important source of effort. Code is written with more care and creativity, because developers are working only on things for which they have real passion [87]. Another reason for the justification of better quality is reusability, which not only helps improve quality but also provides three more benefits. First, since the open source community provides rich base of reusable software at the cost of downloading from the internet, the users can select the best of breed components to reuse in their own system. Second, since the base components are freely available, the project doesn't need to be *created from scratch;* this supports faster system growth. Third, software reuse reduces the development cost. There is no need to spend money on the software, which is already available free of cost. Open source software community claims that open source software possesses greater modularity than closed source software [56]. The rationale is that since open source projects are built to be extended, the modules are developed with minimal coupling else it would be difficult to incorporate changes for system growth [95]. Because open source development is globally distributed, well- defined interfaces and modularized source code are a prerequisite for effective remote collaboration [41].

2.2 Disadvantages of Open Source Software

There have been many claims about quality of the open source software but quality varies from product to product, and there is no standardized process and metrics in existence for quality assessment [123]. No risk assessment is ever performed and no measurable goals are set during open source development [124]. Having no contractual deadlines can be the problem for organizations relying on open source projects as a platform for software development [41].

Open source contributors tend to be more interested in coding than documenting or testing [41]. Some attempts to address the problem of documentation are Linux Documentation Project (www.tldp.org) and Mozilla Developer Documentation Web page (www.Mozilla.org/docs) but these solutions are for large established projects. For small projects it is rare. Testing strategies are not defined, and if any exist they are implicit and are not visible outside the project's developer community. Adopting open source development practices can make organizations pay less attention to strategic planning, detailed requirement elicitation and organized support [41].

Visibility of software architecture is often neglected in open source projects. When the core system is launched on the internet, the details of architecture are not usually specified. It might be available or not. Unintentionally unavailable software architecture suggests that the structure exist in some people's mind only. As the project grows and distributed developers take part in defect reporting and code development this problem becomes more complicated. More emphasis is placed by central organization towards adding new features and the architecture of the system becomes more complex to document and in most of the cases it is not provided even in the mature OSS projects.

Another important issue is the software license associated with the product. If a product is developed by incorporating or using open source product and its reproduced or sell the product without the licensor's permission, the licensor might claim for damages or force you to end the product's production, delivery and sale [115].

Open source software is frequently licensed under GNU GPL (General Public license), which has a so-called "viral" effect: any application incorporating components licensed under GNU GPL [97] must make its entire source code freely available [110]. Those

organizations who want to incorporate OSS components in their product but don't want to make their own product to be OSS are forced to do so.

One of the charms of using open source components is to cut the costs but the consequences of selecting the wrong component can erase these benefits [99]. A very careful attention is required while choosing the OSS components. Factors like maturity, flexibility and longevity are important to be taken into account. The statistics from Sourceforge.net (one of the large platform for OSS) indicate that only five percent of the OSS projects reach to the mature stage [65]. Again there is an issue that if people were relying on OSS and the projects dies than the time and effort for those who were using that product would be wasted. The evolution and stability of OSS communities and OSS systems are mutually dependent [66]. Many open source products have no clear community and are based on single user [41]. "Successful" projects are well-publicized to gain the attraction of developers while practically the large amount of projects die because they depend on one or few core developers and fail to get any sufficient attention. Attracting users is a difficult task because open source developers work on projects that they consider important and significant additions to the software universe. They are not interested in products that would lead to a dead end or would make a small or marginal impact [65]. Even the life of mature projects is at risk. There are number of causes that can lead mature projects towards decline like loss of identifying personalities, influence of profit-oriented companies and too many constraints on developers [68].

3. *Mozilla*- An Open Source Product

One of the projects running under the banner of open source development is Mozilla, a web browser. Netscape announced Mozilla in January 1998. Both the browser and the

source code is available free of charge. During the early years of its release it did not receive as much development effort from outside Netscape as the Mozilla founders were expecting [87]. The reasons behind were large source code size, cumbersome architecture, and absence of working product, poor management and license requirement for the proprietary Motif library [87,88]. One of the project leaders, named Jamie Zawinski resigned, during the second year of Mozilla's launch and described the reasons of quitting are poor management and missed opportunities. Mozilla failed to launch a production-quality browser for two and a half years after the project was launched. This situation began to improve by the end of 2000. The improved documentation on Mozilla architecture and technology (how to build and test the product), tutorials, refined processes and development tools became key elements. The interest in Mozilla is growing and the use of Mozilla development tools in commercial products is exercised by recognized companies (Hewlett Packard, Oracle, Red Hat, and Sun Microsystems), which indicates high quality and scalability of Mozilla [87].

Mozilla is expanding and has announced a new web browser named Mozilla Firefox. At the time of this writing FireFox has 44 million users and it provides the features like popup blocking, tabbed browsing, built in google, live bookmarks and fast download. Other products from Mozilla.org are Thunderbird, an email software, which provides features like junk email filtering, high level security and option to customize toolbar; and Camino, a web browser for the Mac OS X operating system. Mozilla offers a full suite of integrated internet applications including a web browser, e-mail client, address book, web page composer, and chat software and calendar application. Mozilla also provides set of tools in order to manage and develop the projects. The tools provided by Mozilla are [88]

- Bugzilla: An automated bug tracking system, which provides a platform to report and fix the bugs associated with Mozilla web browser.
- Bonsai: A web interface that maintains the log of checkins and is capable of providing checkin information between certain dates, files or by certain developers.
- Tinderbox: A web tool that builds, tests and reports the Mozilla application suite on a 24/7 basis.
- LXR: LXR is a cross-referenced display of Mozilla source code and provide an up to date look of modules from the main Mozilla.org CVS server.

Mozilla is currently operated by Mozilla.org staff, for coordinating and guiding the project and perform some coding [88]. Their roles are in diversified areas like quality assurance, development, product releases and maintenance. Mozilla community has identified its community members into two different groups: Developer network and user community. The Developer network includes Developer forums (mailing lists, newsgroups and chat servers), mozdev.org (project hosting site related to Mozilla and is currently hosting over 150 projects) and Mozilla-qumi (Japanese developer network).

User community provides support to English users and International users. Mozilla English community site provides list of communities working for Mozilla (example: MozillaZine, Mozilla Thunderbird help, Netscape DevEdge) and International communities offer support for over 60 languages.

The current roadmap of Mozilla (which is third roadmap revision) specifies four key elements [89].

- Focus development effort on new standalone applications (FireFox & thunderbird).

- Performing update (especially security update) on application suite's final stable branch (1.7.x) for maintenance.
- Fix Gecko layout architectural bugs for improved performance and extension.
- Moving away from large ownership model to individual ownership of software modules.

Most of the Mozilla source code is written in C++ and java script. Mozilla C++ source code is intended to follow the rules of OOP that includes building modular components. The Mozilla "platform" is composed of a set of technologies and components that can be used to build cross-platform applications [119]. The description of the technologies and components that constitutes core Mozilla architecture is listed on [64]

XPCOM: XPCOM (cross-platform component object model) allows Mozilla to export interfaces and have them automatically available to Javascript scripts. In Mozilla, components exist as singleton service or object instance. Singleton service is an instance of object that is created only once where as object instances can be initiated more than once. The components are written as classes that implement a clearly defined interface. It strongly separate interfaces from implementation thus provides basis for modularity in Mozilla source code. The details of implementation class are hidden and the client only interacts with interface class. Due to the potential of XPCOM with dealing interfaces, objects could be created in other languages too for example Javascript. The XPCOM uses XPIDL (A cross-platform interface definition language) to generate C++ header files for XPCOM objects.

XPConnect : It provides service for javascript objects to access XPCOM objects by building a wrapper around objects or vica-versa. The main goal is to provide object

transparency on either side of XPCOM interface. Mozilla source code is written in C++ and java script. C++ is a compiled language where as JavaScript is an interpreted language. Components written in C++ starts at their own, where as to run JavaScript scripts XPConnect is used. JavaScript is written mostly for user interface events. In order to use XPConnect the XPCOM interface must be developed in XPIDL.

XUL: XUL is used to specify user interface appearance and application logic. The combination of appearance and behavior is called "chrome", which can be loaded from .xul files and associated JavaScript and CSS files. The code in .xul files is capable of changing browser behavior. The cross platform installations, packaging and software update are performed by XPInstall technology.

XPInstall: XPInstall is a technology for performing cross-platform installations, packaging, and software updates.

NSPR: Netscape Portable Runtime (NSPR) is a cross platform neutral API for operating systems [88]. As discussed earlier most of the code is written in C++ and java script. C++ is a portable language but its portability is limited to program logic and data structures only. For using Mozilla components across platforms NSPR provides a layer between the OS and the Mozilla source code. This allows simpler coding in other areas of the Mozilla source code.

The contents of this chapter were addressed to in general open source software systems and specific to Mozilla. This discussion was desired because we have conducted the research on Mozilla, which is large-scale open source software system. The advantages and disadvantages of using OSS systems are described in detail. One of the disadvantages associated with OSS systems is lack of documentation.

Since we have conducted a KDD project and the first step in performing KDD process is to understand application domain. In order to understand the defect-tracking database of Mozilla web browser (Bugzilla), we have performed database reverse engineering to develop the conceptual model of database using entity-relationship (ER) diagram. The next chapter is addressed to the details of database reverse engineering along with the description of Bugzilla database and development details of its ER diagram with ER diagram itself.

CHAPTER 4

1. Introduction

The detailed overview of open source software and its advantages and disadvantages were discussed in chapter 3. One of the disadvantages associated with open source projects is lack of documentation. In order to construct the data set based on Mozilla source code along with its defect tracking system (Bugzilla); we needed to have an abstract model of Bugzilla database, which was not available at that time from Mozilla. We have performed data reverse engineering on Bugzilla database to develop the entity relationship diagram.

This chapter provides the introduction of data reverse engineering and entity relationship diagram and then a detailed overview of Bugzilla database. The last section of this chapter is the entity relationship diagram of Bugzilla database, which is performed as a part of conducting this research.

2. Data Reverse Engineering

Data Reverse Engineering (DRE) can be regards as adding value to existing data assets, making it easier for organizations to use and more effective as a tool [92]. It is the process of analyzing the stored data, identifying and extracting the desired information. The increased use of data warehouses and data mining techniques for strategic decision support systems have also motivated an interest in data reverse engineering technology.

Data reverse engineering process consists of two major activities that are data analysis and abstraction respectively. Data analysis is a human intensive exploratory activity. It is performed to obtain an up to date logical data model of the target database since in most of the cases information about the logical model is missing in the physical schema. The

logical data model describes details about data structure. Conceptual abstraction maps logical data model into conceptual design. Conceptual design is the formal specification of user requirements presented in an abstract form and is usually represented either by entity relationship diagram or object –oriented model.

A brief overview of data reverse engineering is given in [109]. By mid 80's Nilson and Davis, in two separate papers [93,49] proposed a translation algorithm for COBOL data structures. By late 1980's Davis and Arora [50] proposed the conversion of a relational Database Model into an entity relationship model. Another attempt during that period was by Boulanger and March [24] for analyzing the information content of existing databases.

In early 1990's the paper by Hainaut [55] provided the details of DRE techniques and strategies. Winans and Davis [103] also addressed the issue of performing DRE. By mid and late 1990's DRE got a significant attention. P. Aiken's [98], M. Blaha's [9] and Dayani-Fard and Jurisca [25] discussed DRE as knowledge retrieval technique.

The advantages offered by DRE are numerous. DRE techniques can be used to assess the overall quality of the software systems [92]. An implemented persistent data structure with significant design flaws indicates a poorly implemented software system. It can also be used to assess the quality of DBMS schema of vendor software, and thus it can represent one of the evaluation criteria for a potential software product [9]. The increased use of data warehouses and data mining techniques for strategic decision support systems have also motivated an interest in data reverse engineering technology [92]. DRE is also being used to extract business constraints that are unknown and hidden within hidden legacy system [98]. This is proving to be very valuable in the area of system

maintenance. Data reverse engineering can also elucidate poorly documented existing software, when developers are no longer available for advice. It also helps in upgrading past hierarchical, network, and relational database to modern relational and object – oriented databases [109].

3. Entity Relationship Diagram

The ER diagram is a semantic data-modeling tool, which is used to accomplish the goal of abstractly describing or portraying data [3]. As the name implies ER diagram contains entities (tables), which are related to each other via some relationships. These relations are based on key attributes, which are actually data items that describe an entity. Primary and foreign keys are the most basic components on which relational theory is based. The Primary key uniquely identifies each instance of the entity while the foreign key is used to refer to instances of other entities. Primary key is usually represented by a single column however sometimes it is created on multiple columns too. To qualify as a primary key for an entity, an attribute must have the following properties:

- It must have a non-null value for each instance of the entity
- The value must be unique for each instance of an entity
- The values must not change or become null during the life of each entity instance.

Whereas the criteria for qualifying foreign key are

- It must be a primary key for a different entity.
- It allows null and duplicate values.

Cardinality defines the numeric relationships between occurrences of the entities participating in the relationship. They describe both directions of the relationship. The appropriate mapping cardinality for a particular relationship set depends on the real world

being modeled. There are four basic relationships that can be used to describe the relationship of one table to another. For example if we consider two entities A and B, then the mapping cardinalities would be

One-to-one: An instance in A is associated with at most one instance in B, and an instance in B is associated with at most one instance in A

One-to-many: An instance in A is associated with any number of instances in B. An instance in B is associated with at most one instance in A.

Many-to-one: An instance in A is associated with at most one instance in B. An instance in B is associated with any number of instances in A.

Many-to-many: Instances in A and B are associated with each other in any number.

There are few more subsets possible for the above general cases of cardinality. But we have focused only on general ones.

4. Bugzilla Overview

Chapter 3 provided the overview of open source software and Mozilla organization. Netscape launched Mozilla in January 1998. The browser and the associated code were published on web free of charge. The group mozilla.org acts as a central point of contact. It supports many technologies including development tools CVS, Bugzilla, Bonsani and Tinderbox. These tools are not the part of the web browser [88].

The defects associated with the web browser are reported and tracked via the platform of Bugzilla, which is a defect-tracking tool. Defect tracking tools are used for conducting defect-tracking procedure. Defect tracking tools are based on databases so that the data stored in databases about software bugs could be accessed and retrieved in almost any

fashion, according to the criteria set by the user. Bugzilla allows individual or groups of developers to keep track of outstanding bugs (defects) in their products. [16].

5. Bugzilla Database

We have used Bugzilla database instance and Mozilla source code to develop data set. Since Mozilla is an open source product. one of the problem related to OSS is lack of documentation .The Bugzilla database contains 34 tables and it was quite complex to understand without having any conceptual model. To overcome this problem, we performed data reverse engineering for developing entity relationship diagram of Bugzilla database from its source code. The data definitions for each table were observed for the identification of attribute name, type and cardinality. The description of attributes and identification of their cardinal relationship is very important for understanding and extracting data from any database. The development of entity relationship diagram is not only useful for conducting this research but is also a contribution for users of Bugzilla.

Bugzilla was written by Terry Weissman in a programming language called TCL and supported by a robust RDBMS at the backend. It was later ported on perl with a CGI web GUI and MySQL is used to store the tables. Bugzilla is a MySQL relational database, and is comprised of 34 tables.

The data types used in Bugzilla are:

- Char: A fixed-length string that is always right-padded with spaces to the specified length when stored. The maximum range is 0 to 255 characters.
- Varchar: A variable-length string in which trailing spaces are removed when the value is stored. The range is from 0 to 255 characters.

- Text: A BLOB or TEXT column with a maximum length of 65535 (2^16 - 1) characters.
- Mediumtext: A BLOB or TEXT column with a maximum length of 16777215 (2^24 - 1) characters.
- Tinytext: A BLOB or TEXT column with a maximum length of 255 (2^8 - 1) characters.
- Int: A normal-size integer. The signed range is -2147483648 to 2147483647. The unsigned range is 0 to 4294967295.
- Mediumint: A medium-size integer. The signed range is -8388608 to 8388607. The unsigned range is 0 to 16777215.
- Smallint: A small integer. The signed range is -32768 to 32767. The unsigned range is 0 to 65535.
- Tinyint: A very small integer. The signed range is -128 to 127. The unsigned range is 0 to 255.
- Datetime: A date and time combination. The supported range is '1000-01-01 00:00:00' to '9999-12-31 23:59:59'. MySQL displays DATETIME values in 'YYYY-MM-DD HH:MM:SS' format, but allows you to assign values to DATETIME columns using either strings or numbers
- Timestamp: A timestamp. The range is '1970-01-01 00:00:00' to the year 2037.
- Decimal: An unpacked floating-point number. Behaves like a CHAR column.
- Enum: An enumeration. A string object that can have only one value, chosen from the list of values 'value1', 'value2', ..., NULL or the special "" error value. An ENUM can have a maximum of 65535 distinct values.

The keys used in Bugzilla database are:

- P: Stands for Primary Key
- M: Represents that the field is a part of a multiple column index
- U: Describes Unique index

6. Entity Relationship Diagram of Bugzilla Database

As described earlier, the Bugzilla database is comprised of 34 tables. The referential integrity maintained between these tables is via references. References serve the same purpose as foreign keys, but they don't appear in the structure of the database. The rationale behind using references is, the time Bugzilla was developed MySQL didn't support foreign keys. It does now (via the InnoDB table type), but Bugzilla still hasn't been modified to use them. Another important feature to note is, none of the primary key are based on multiple columns i.e. there are no composite primary keys.

The observation of ER diagram shows that most of the mapping cardinalities are one-to-many or none however other types of mapping cardinalities also exist like many-to-one, one-to-one, one-to-one or none and many-to-one or none. Figures [4.3, 4.4, 4.5,4.6, 4.7, 4.8] are the entity relationship diagrams of Bugzilla database.

The observation of figures 4.1, 4.2, 4.3, 4.4, 4.5, 4.6 shows that there are eight attributes (columns) responsible for establishing cardinality in the schema. They are userid (profiles), bug_id (bugs), id (groups), id (products), id (flagtypes), id (components), attach_id (attachments) and fieldid (fielddefs). The description of these attributes is given below.

- **userid.profiles** :uniquely identifies a user via numeric value.
- **bugs_id.bugs:** unique identification of a bug.

- **id.products:** identify product type for which the bug is reported.
- **id.groups** : identifies the group of users.
- **id.flagtypes:** defines flagtypes.
- **id.components** : used for the identification of individual components.
- **attach_id.attachments** : represents a patch.
- **field_id.fielddefs** : uniquely identifies the fields.

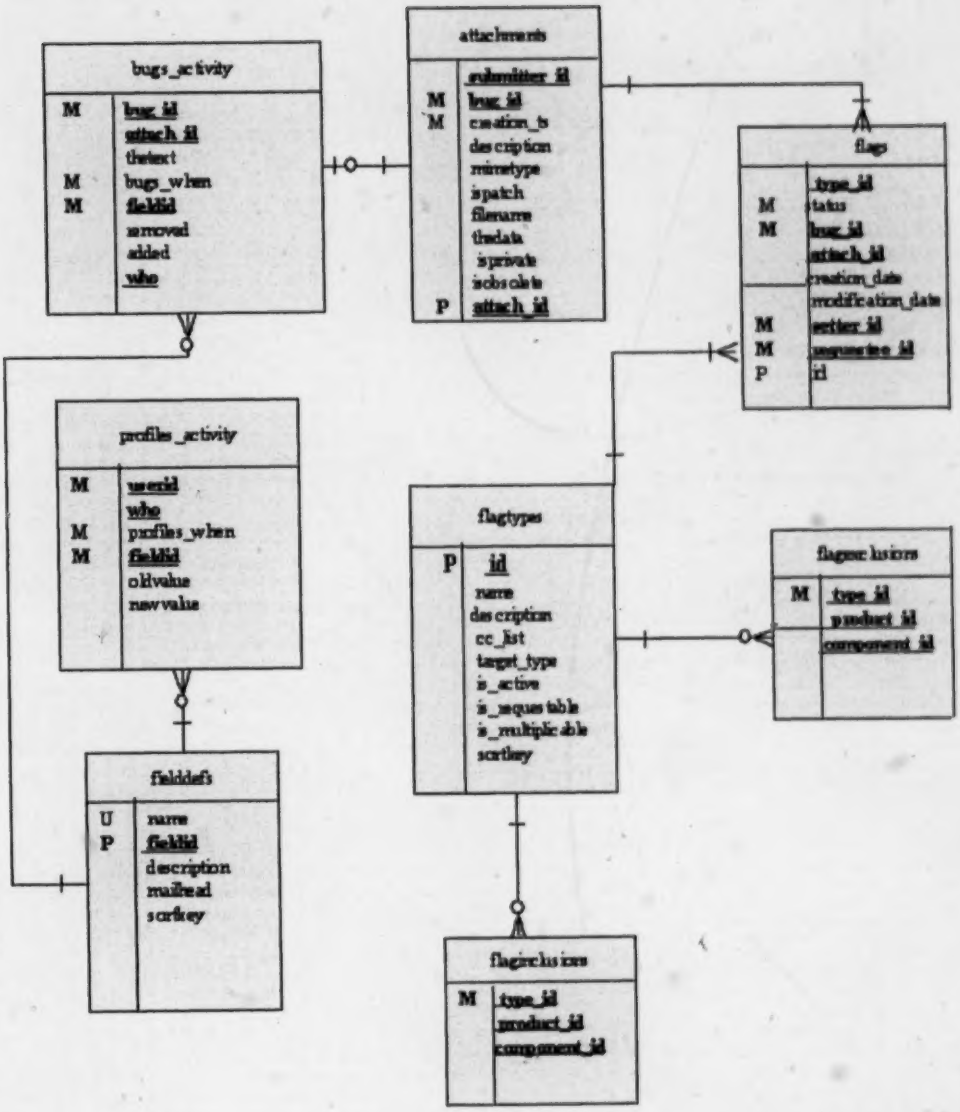

Figure 4.1: Entity Relationship Diagram (page 1)

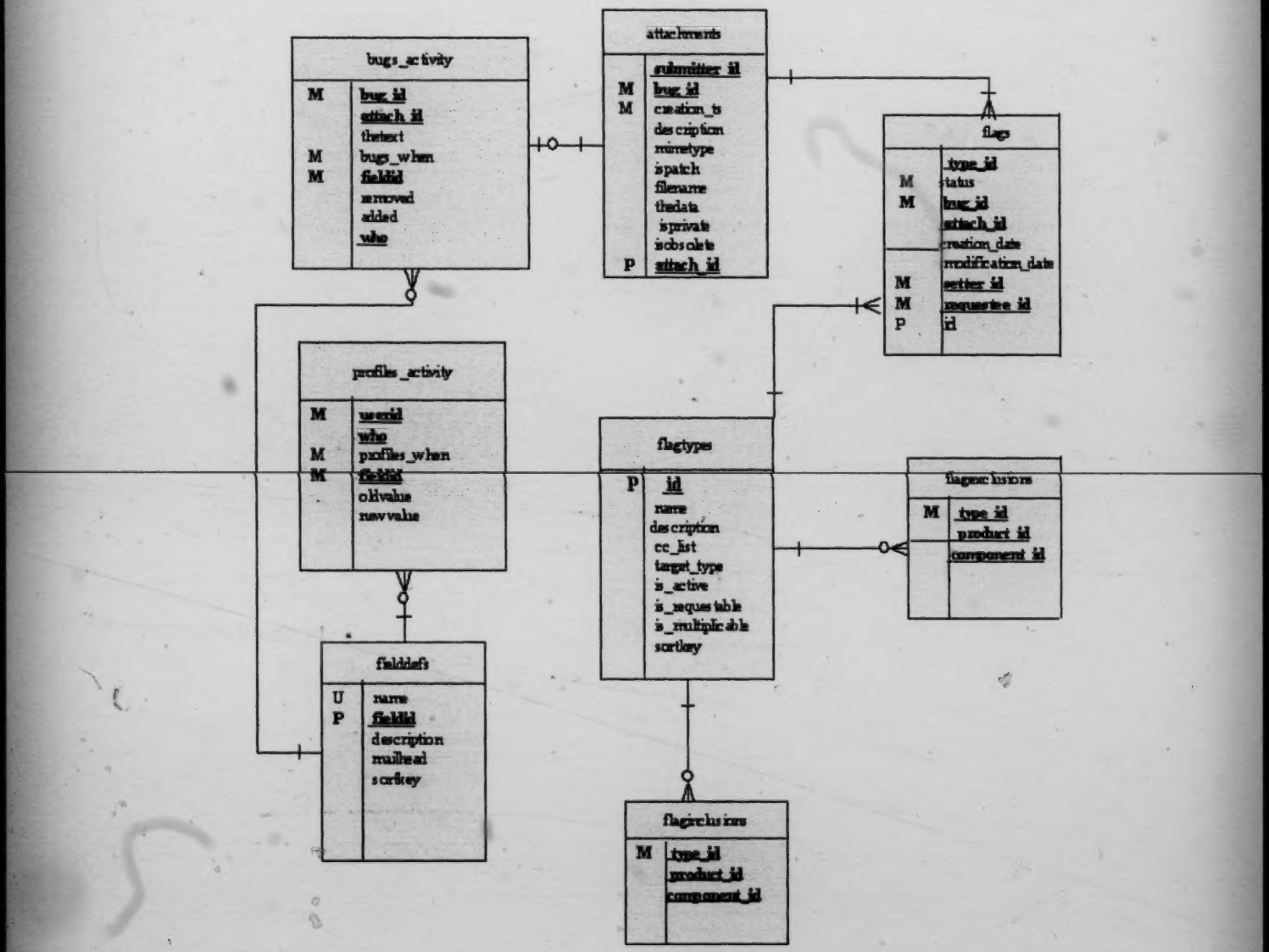

Figure 4.2: Entity Relationship Diagram (page 2)

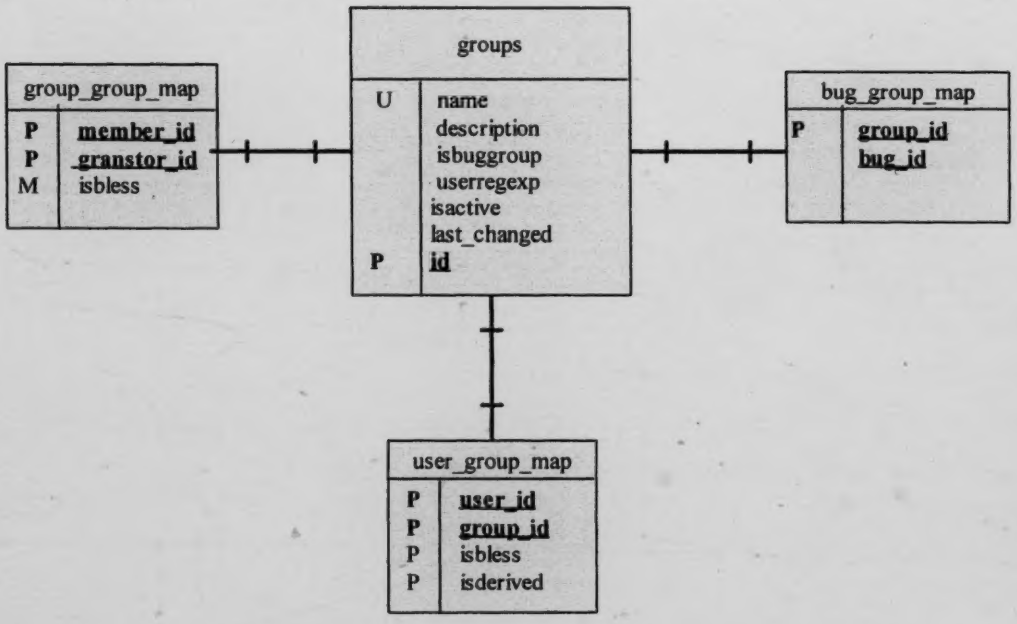

Figure 4.3: Entity Relationship Diagram (page 3)

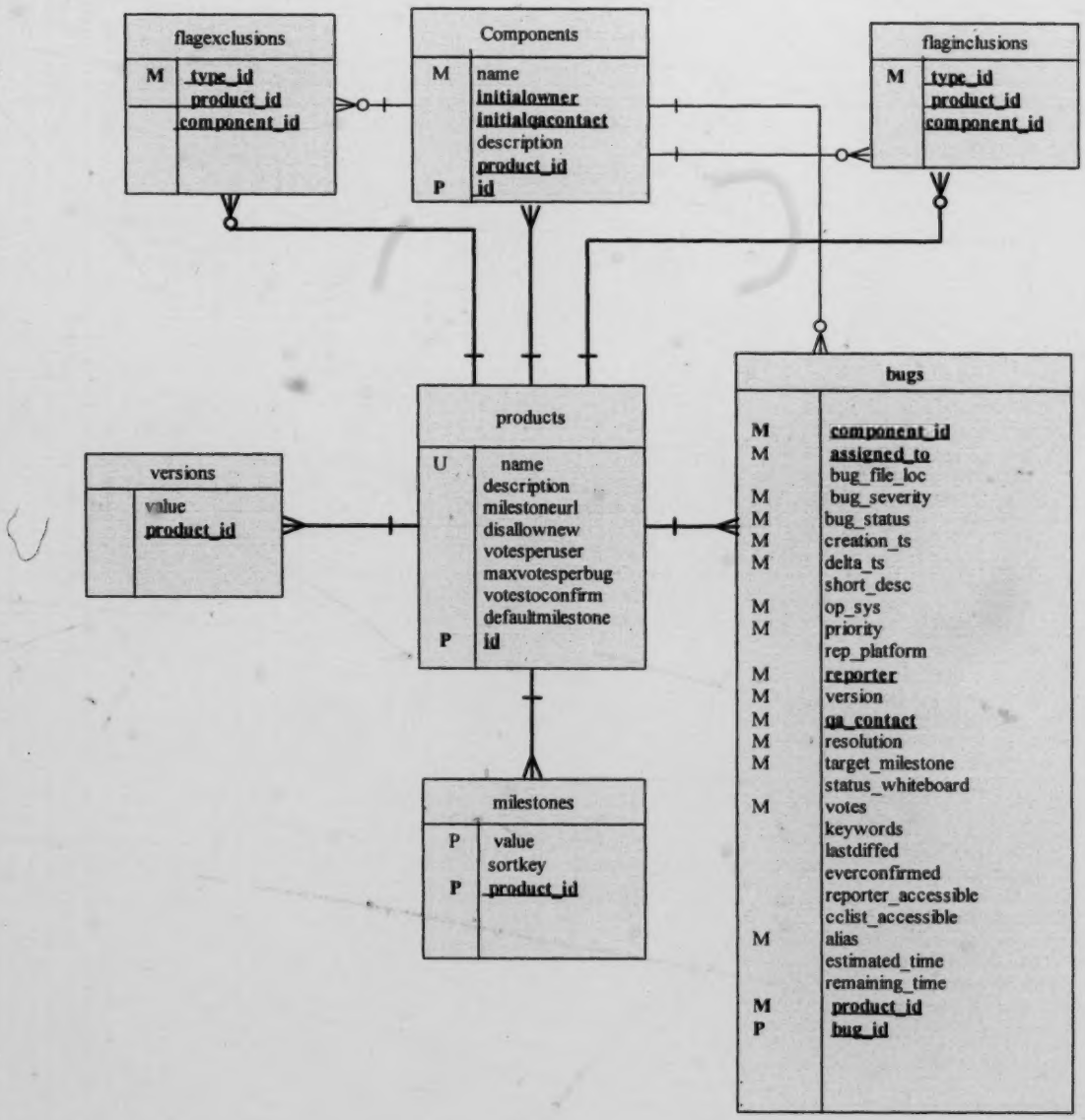

Figure 4.4: Entity Relationship Diagram (page 4)

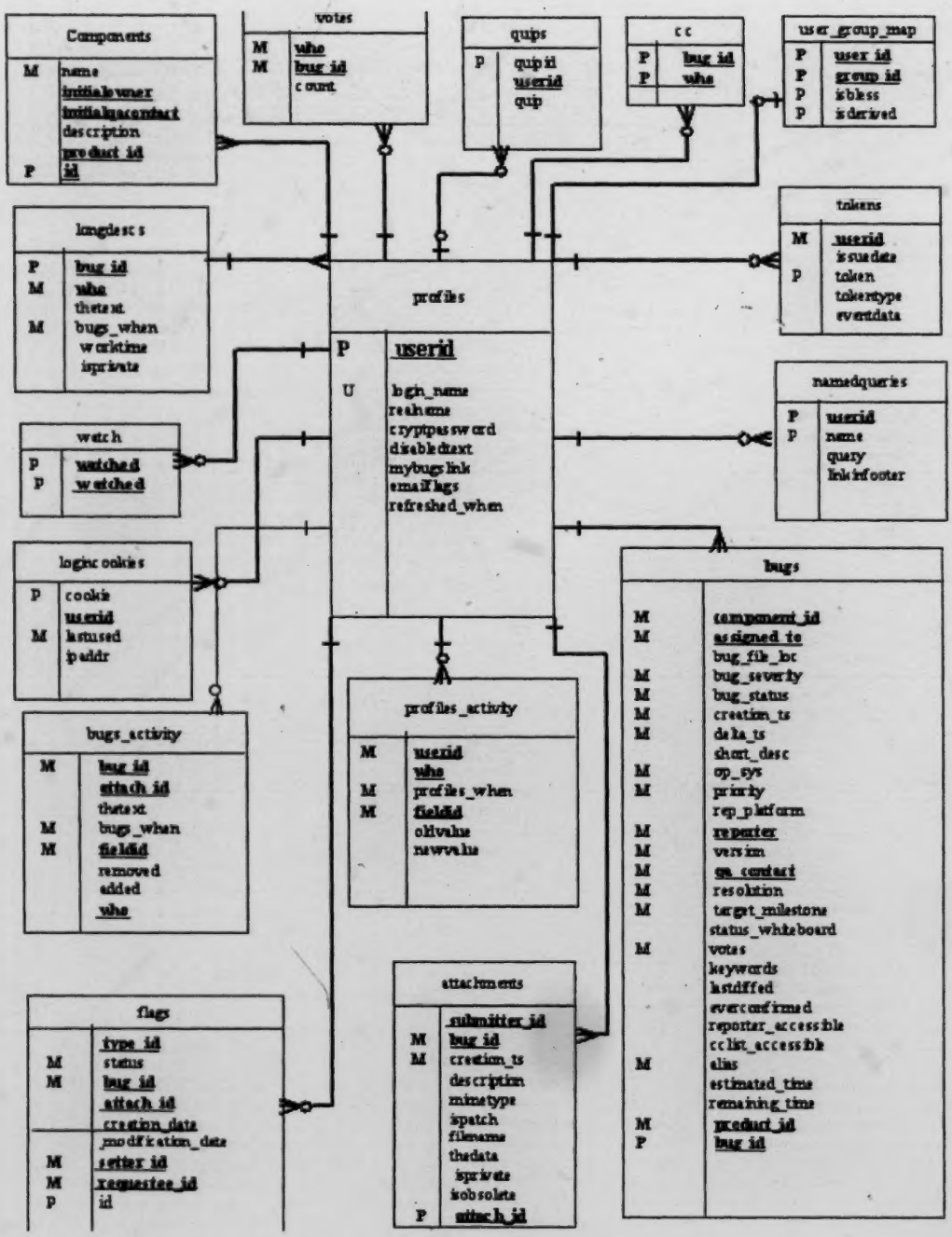

Figure 4.5: Entity Relationship Diagram (page 5)

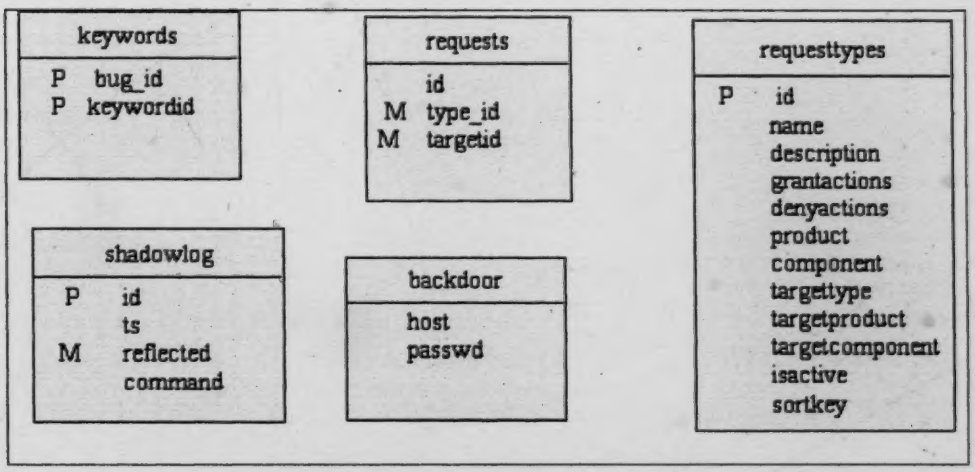

Figure 4.6: Entity Relationship Diagram (page 6)

The ER diagram helped in understanding Bugzilla database and identifying the required data. The description of desired data and data extraction process, which is used for the development of data set are provided in the next chapter. Data preparation is the most resource intensive stage of conducting any KDD project and consumes up to 60-70 % of the total effort [59]. The following chapter is dedicated to the details of development of Metrics-Delta data set developed from Mozilla source code and Bugzilla database.

CHAPTER 5

1. Introduction

Data Preparation is the most resource intensive stage [59] of the data mining. The research on analyzing the data mining process suggests that almost 60% to 70 % of the total effort is dedicated to the data preparation. The rationale behind is quite logical. All data mining projects are domain specific and every domain has its own specific characteristics, which comes with its typical problems like noise, incompleteness, complexity and volume. Along with these data specific problems, the important question is among large volumes, what specific data is of point of interest. The extraction of data depends on the objective of the data mining process, which illustrates what information is required and what would be the use of that extracted information.

The main goal of our project is mining software quality data from a large-scale open source product to develop predictive model. The data set used to conduct this project is developed using source program of *Mozilla* web browser release 1.1 and an instance of its associated defect-tracking database *Bugzilla.* Mozilla is an open source code product that is its source code can be downloaded freely whereas the instance of Bugzilla was generously provided by Mozilla organization for conducting this academic research.

We aimed to develop a data set by collecting OO metrics for all C++ classes present in Mozilla source code along with the defect density of each class. The presence of defective modules posse considerable risk on software quality, however the effective risk prediction models can improve software developer's ability to identify defect-prone modules and focus quality assurance activities on those modules. The defect density of

any module can be measured by "Delta". The value of delta can be very useful for predicting software quality. Audris M. [87], has defined delta as a defect density that is the number of times changes are applied to the code due to reported problems. In [87], he has used the value of delta to compare the software quality of two large scaled open source software systems Apache web server and Mozilla web browser with commercial software systems. Since we aimed to identify the value of delta for every C++ class present in Mozilla source code and in order to do that it was required to find the value of delta for each .cpp and .h file. After the determination of file level delta the next step was to assign it to the class level according to source lines of code for each class and its associated methods.

The measures of defect density along with software metrics are useful quality predictors. The use of software metrics to predict software quality is a well-known approach. Measures obtained from the source code such as complexity, coupling, and cohesion have been associated with the risk factors such as defects and change. Khosguftar T.M, [132] has proposed the use of object-oriented metrics along with defect counts to predict software quality using a KDD process. The importance of object oriented metrics and benefits provided by using these metrics are already discussed in detail in chapter 1.

Software metrics for all C++ classes in Mozilla source were collected using metric tool "Krakatu Professional for C++ ". Krakatau provided "22" attributes for each class including classical CK metrics. The final data set, which contains delta value and 22 metrics for each class is used for conducting prediction experiments (see chapter 6).

The data set is called *Metrics-Delta data* set and is developed in three steps.

1. Creation of Delta Data set
2. Creation of Metrics Data set
3. Joining Delta data set with Metrics data set to obtain Metrics-Delta data set.

For the development of Delta data set and for joining Metrics data set with delta data set, the programs are written in *Perl*. Perl is a free and powerful language. For our interest, Perl offers a built-in support for pattern matching. A pattern could be searched among large amounts of data by creating a regular expression for that particular pattern. Along with the strong pattern matching capabilities another important feature of Perl is its broad range of nested data structures like Arrays of Arrays, Hashes of Arrays, Arrays of Hashes, Hashes of Hashes and Hashes of Functions. We have used both Perl's pattern matching capabilities and flexible data structures for the development of data set.

This chapter describes in detail the process of creating Delta data set and Metrics data set followed by the details of creating the Metrics-Delta data set. Assignment of defects from file-level to individual modules was a difficult task. In conventional C++ programming approach one class is defined in one file (header file). But in Mozilla source code a file may contain definitions of more than one class. The defect count of one file has to be assigned according to number of classes in a file. Similarly, the methods of class are defined in CPP files and conventionally methods of only one class are defined in one CPP file. Again in Mozilla methods from more than one class were defined in one file.

It was necessary to map our file-level defect counts to the class-level metrics collected and so we elected to divide our file-level defects according to the fraction of lines of program devoted to a class in given file. This point will be further discussed in section 4 of this chapter.

2. Creation of Delta Data Set

As discussed earlier we aimed to identify the delta value for each C++ class present in Mozilla source code using instance of Bugzilla database, the first step in accomplishing this task was to calculate value of delta for file. In chapter 4, the details of performing data reverse engineering on Bugzilla database is provided. The analysis of the entity relationship diagram helped in identifying the desired attributes. For the population of Bugzilla database, we have taken into account the following attributes from two tables "bugs" & "attachments" (see Figure 4.3).

- *bug_id*: a unique identification number associated with every reported bug in bugs table.
- *resolution*: identifies the type of resolution (Fixed, Invalid, Wontfix , Duplicate, moved & worksforme)
- ispatch: a binary attribute that identifies attachments as a patch
- thedata: the description of all the files that were patched along with the patch itself.

Once required attributes were identified, the next step was performing MySQL queries on the database to extract the required data. The database instance contains 190722 reported bugs. The "bugs" table has 28 attributes, which stores data related to the reported bugs. One of the attributes among these 28 attributes in bugs table is the "resolution" of the

bugs. The data type in the resolution field is predefined that is it should posses one of the following value that are "Fixed", "Invalid", "Wontfix", "Duplicate", "Moved" & "Worksforme". Since we were interested in *fixed* bugs only, with the help of a SQL query all the fixed bugs were extracted and it showed that out of 190722 reported bugs only 52252 were fixed.

The next question was to figure out among fixed bugs, which bugs were fixed using patches? This question is answered by examining the "attachments" table [Figure 4.3]. The "attachments" table contains a field named "ispatch" [Figure 4.3]. This a binary attribute and determines that whether a bug is fixed by applying a patch or not. Before observing this variable, we assumed that every bug that is fixed by applying a patch on the software module. However, we found this was not the case. We contacted Bugzilla organization to resolve this problem. The answer provided by the bugzilla developer's forum was " Every rule has an exception too", occasionally, a bug will be fixed without a patch because the solution was trivial or the module it was touching doesn't require peer review or it is also possible that somebody selected a wrong resolution [75]. There is also no enforcement mechanism mandating that a patch exists in Bugzilla before it can be checked-in into CVS.

The query results from attachments table showed that among 52252 fixed bugs only 4665 bugs were fixed using patches. The number of bugs fixed using patches is less than ten percent of fixed bugs. The variable "thedata" in the "attachments" table contains

the description of all the files that were patched along with the patch itself. The path information of the patched file is given by "RCS file: //path" in this attribute. With the

help of MySQL statements all the patches of software files were extracted in a new table and were exported out of the database as a text file [see Figure 5.1].

```
RCS file: /cvsroot/mozilla/xpcom/threads/nsEventQueueService.h,v\
retrieving revision 1.10\
diff -u -b -w -r1.10 nsEventQueueService.h\
--- nsEventQueueService.h\ 2000/03/04 03:16:49\       1.10\
+++ nsEventQueueService.h\ 2000/06/01 07:39:10\
@@ -29,18 +29,6 @@\
 class nsIEventQueue;\
 class EventQueueEntry;\
-// because available enumerators can't handle deletions during
enumeration\
-class EventQueueEntryEnumerator {\
-public:\
-  EventQueueEntryEnumerator();\
-  virtual ~EventQueueEntryEnumerator();\
-  void              Reset(EventQueueEntry *aStart);\
-  EventQueueEntry *Get(void);\
-  void              Skip(EventQueueEntry *aSkip);\
-private:\
-  EventQueueEntry *mCurrent;\
-};\-\ class nsEventQueueServiceImpl : public nsIEventQueueService\
@@ -76,13 +64,12 @@\ Addref the descriptor in any case. parameter aNative
is\ignored if the queue already exists. */\NS_IMETHOD
CreateEventQueue(PRThread *aThread, PRBool aNative);\
+  NS_IMETHOD MakeNewQueue(PRThread* thread, PRBool aNative, nsIEventQueue
**aQueue);\
   void       AddEventQueueEntry(EventQueueEntry *aEntry);\
   void       RemoveEventQueueEntry(EventQueueEntry *aEntry);\
-  nsHashtable     *mEventQTable;\
-  EventQueueEntry *mBaseEntry;\
+  nsSupportsHashtable *mEventQTable;\
   PRMonitor       *mEventQMonitor;\
-  EventQueueEntryEnumerator mEnumerator;\
```

Figure 5.1: Sample Patch

Figure 5.1 represents a typical patch used to remove the reported defect from Mozilla source code. Each patch contains the name of the file along with its complete path, which is used to extract the value of Delta.

From raw patches the file names were extracted and then were cleaned from undesired characters. Since the objective was to find delta value of C++ classes, all the files with extension .cpp and .h were identified. The file names were than compared against

directory of Mozilla source code to calculate value of delta. The section 2.1 and 2.2 describes in detail how the value of delta is determined.

2.1 Data Extraction

The text file (see Figure 5.1), contained raw text data, which were patches and file names along with complete paths of the patched files. The detailed description of how the value of "Delta" is extracted from data in the text file [Figure 5.1] is given below.

The program [Program 5.1] was executed on the text file [Figure 5.1] recognized all " RCS file: " patterns and wrote them to another text file [Figure 5.2].

```
RCSfile:/cvsroot/mozilla/xpcom/threads/nsEventQueueService.cpp,v\
RCS file:/cvsroot/mozilla/xpcom/threads/nsEventQueueService.h,v\
RCSfile:/cvsroot/mozilla/mailnews/imap/src/nsImapProxyEvent.cpp,v\
RCS file:/cvsroot/mozilla/mailnews/imap/src/nsImapProxyEvent.h,v\
RCS file:/cvsroot/mozilla/profile/Acct/nsAccount.cpp,v\
RCSfile:/cvsroot/mozilla/xpfe/appshell/src/nsWebShellWindow.cpp,v\
```

Figure 5.2: Extracted Patched File Names

The file names were cleaned from undesired characters by developing second program [Program 5.2]. Since the Perl program extracted complete line started with the patterns

```
RCS file:/cvsroot/mozilla/xpcom/threads/nsEventQueueService.cpp
RCS file:/cvsroot/mozilla/xpcom/threads/nsEventQueueService.h
RCS file:/cvsroot/mozilla/mailnews/imap/src/nsImapProxyEvent.cpp
RCS file:/cvsroot/mozilla/mailnews/imap/src/nsImapProxyEvent.h
RCS file:/cvsroot/mozilla/profile/Acct/nsAccount.cpp
RCS file:/cvsroot/mozilla/xpfe/appshell/src/nsWebShellWindow.cpp
RCS file:/cvsroot/mozilla/xpfe/appshell/src/nsXULWindow.cpp
```

Figure 5.3: Cleaned File Names

"RCS file: " the line had some additional undesired characters at the end of the path [Figure 5.3]. Since we were only interested in ".cpp" and ".h" file , the next two programs [Program 5.3, Program 5.4] identified .cpp and .h files respectively and send the results to a text file [Figure 5.4].

```
RCS file:/cvsroot/mozilla/xpcom/threads/nsEventQueueService.cpp
RCS file:/cvsroot/mozilla/xpcom/threads/nsEventQueueService.h
RCS file:/cvsroot/mozilla/mailnews/imap/src/nsImapProxyEvent.cpp
RCS file:/cvsroot/mozilla/mailnews/imap/src/nsImapProxyEvent.h
RCS file:/cvsroot/mozilla/profile/Acct/nsAccount.cpp
RCS file:/cvsroot/mozilla/xpfe/appshell/src/nsWebShellWindow.cpp
RCS file:/cvsroot/mozilla/xpfe/appshell/src/nsXULWindow.cpp
```

Figure 5.4: List of patched .h and .cpp Files

2.2 Data Transformation

In our project the desired information was Defect counts per file, we call it "Delta"- the number of atomic changes in a file. The details of obtaining the value of Delta for all .cpp and .h files in a Mozilla source program are given below.

```
/mozilla /xpcom/threads/nsEventQueueService.cpp
/mozilla /xpcom/threads/nsEventQueueService.h
/mozilla /xpcom/threads/nsEventQueueService.o
/mozilla /xpcom/threads/nsEventQueueUtils.h
/mozilla /xpcom/threads/nsIEventQueue.idl
/mozilla /xpcom/threads/nsIEventQueueService.idl
/mozilla /xpcom/threads/nsIProcess.idl
```

Figure 5.5: Mozilla Directory Sample

First the names of all files of Mozilla source program along with their complete paths were generated by writing and executing the program [Program 5.5] and saved in a text

file [Figure 5.5]. The two following programs [Program 5.6, Program 5.7], extracted all .cpp and .h files from a text file [Figure 5.6] and the list of final desired files was written to the text file [Figure 5.6].

```
/mozilla /xpcom/threads/nsEventQueueService.h
/mozilla /xpcom/threads/nsEventQueueUtils.h
/mozilla /xpcom/threads/nsPIEventQueueChain.h
/mozilla /xpcom/threads/nsProcess.h
/mozilla /xpcom/threads/nsEventQueueService.cpp
/mozilla /xpcom/threads/nsProcessCommon.cpp
/mozilla /xpcom/threads/nsProcessMac.cpp
```

Figure 5.6: List of .h and .cpp Files in Mozilla Directory

We had now two text files. First [Figure 5.4], which contains names of CPP and header patched files from Bugzilla database instance and second [Figure 5.6], whose contents are names of all CPP and Header files of Mozilla source program. To obtain the value of Delta, another Perl program was developed [Program 5.8], which compared two text files [Figure 5.4] and [Figure 5.6]. The Program performed this comparison by first reading the contents of figure 5.6 sequentially in a loop. It read every file name in a nested loop, searched its occurrence in figure 5.4. The default value for Delta was set as zero and was incremented by one each time the search pattern matches. The execution of the program provided us the defect counts for all the CPP and Header files for Mozilla source Program [Figure 5.7]. We call it a Delta data set, which has two attributes File names and Delta.

```
/xpcom/threads/nsEventQueueService.h
 8
 /xpcom/threads/nsEventQueueUtils.h
0
 /xpcom/threads/nsPIEventQueueChain.h
6
 /xpcom/threads/nsProcess.h
0
 /xpcom/threads/nsThread.h
1
/xpcom/threads/nsEventQueueService.cpp
11
 /xpcom/threads/nsProcessCommon.cpp
0
```

Figure 5.7: Delta Data Set

3. Metrics Data Set

The Metrics data set was developed using Krakatau Professional (C++) [64]. The Krakatau Professional C++ is a software metrics tool designed for collecting software metrics for C++ source Program. In order to create Metrics data set on Mozilla source Program, the following activities using Krakatau Professional were performed.

1. Project creation for Mozilla source code.
2. Applying metrics on created project.
3. Creating Report.

The report named as "Mozmetrics report" contains a comprehensive set of metrics at the level of Files, Classes and Methods. The Krakatau report shows that Mozilla source Program contains 8349 C++ classes and the class-level metrics obtained from Krakatau:

- CBO-Coupling between objects: Calculated by totaling the number of unique types of attributes with in each class. The count only includes user-defined object types. The larger this number, the greater coupling between object classes.
- CSA-Class size Attributes: Number of attributes of a class.

- CSAO-Class size Attributes and Operations: The total number of attributes and operations for a class.
- CSI-Class specialization Index: (NOOC * DIT) / Total Methods.
- CSO -Class size Operations: The number of operations for a class.
- DIT-Depth of Inheritance tree: From the current class traces up to entire inheritance hierarchy counting longest route. The value of DIT is stored against the current class, so for the root class DIT=1, first level subclass DIT=2 and so on.
- LCOM-Lack of cohesion methods: This metric quantifies how much intercourse the methods of this class have with the member variables of that class. It is obtained by subtracting number of similarities between methods from number of non-similarities between methods.
- LOC-Lines of code: Number of lines in this class including source, white space and comments.
- NAAC- Number of attributes added: Provides the count of number of new attributes that a class has in respect to its immediate super-class. For a root class NAAC= CSA.
- NAIC-Number of attributes inheritance: NAIC is a count of number of attributes that a class has inherited from its super-class. For a root class NAIC=0.
- NOAC- Number of Operations Added by a Class: It is a count of new operations that a class has in respect to its immediate super-class. For root class NAOC=CSO.
- NOCC- Number of child classes: Number of sub classes of a root class.
- NOIC- Number of operations inherited by children: Counts of number of operations that a class has inherited from its super-class. For a root class , NOIC=0.

- NOOC- Number of operations in a class: It is a count of number of operation that a class has overridden. For a root class, NOOC=CSO.
- NpavgC-Average number of method parameters: Total parameters / number of methods. This metric if over 10, would indicate that too much information is being passed into methods and that more attributes should be added to the class.
- Osavg- Average operation size = Osavg is given by WMC/ number of methods. It provides measure of the complexity of the class. Classes with Osavg greater than 10 should be perhaps redesigned.
- PA- Private attribute usage: It is sum of all references to distinct private attributes within all methods of a class.
- PPPC-Percentage public/protected members: PPPC is simply the percentage of public and protected members of a class in respect to all members of class.
- RFC-Response for class: It is the number of methods in a class plus number of distinct methods called by those methods.
- SLOC-Source lines of code: SLOC is the number of source lines in a class excluding white space and comments.
- TLOC-Total lines of code: It is the count of LOC for the class and its methods.
- WMC- Weighted methods in class: WMC provides the sum of cyclomatic complexity for all methods in the class. This number reflects the complexity of a class and classes with WMC larger than 100 should perhaps be redesigned.

The Mozmetrics report was converted to a table, in which each record contained a class name (record key), File name and the values of all 22 class-level metrics.

4. Metrics-Delta Data Set

Since Delta data set holds defect counts for individual files and Metrics data set contains metrics at the level of classes, the third step is to create a Metrics-Delta data set, that can combine these two data sets using filename as a key and assign defect counts at the level of classes. This final data set should obtain the form of Figure 5.8.

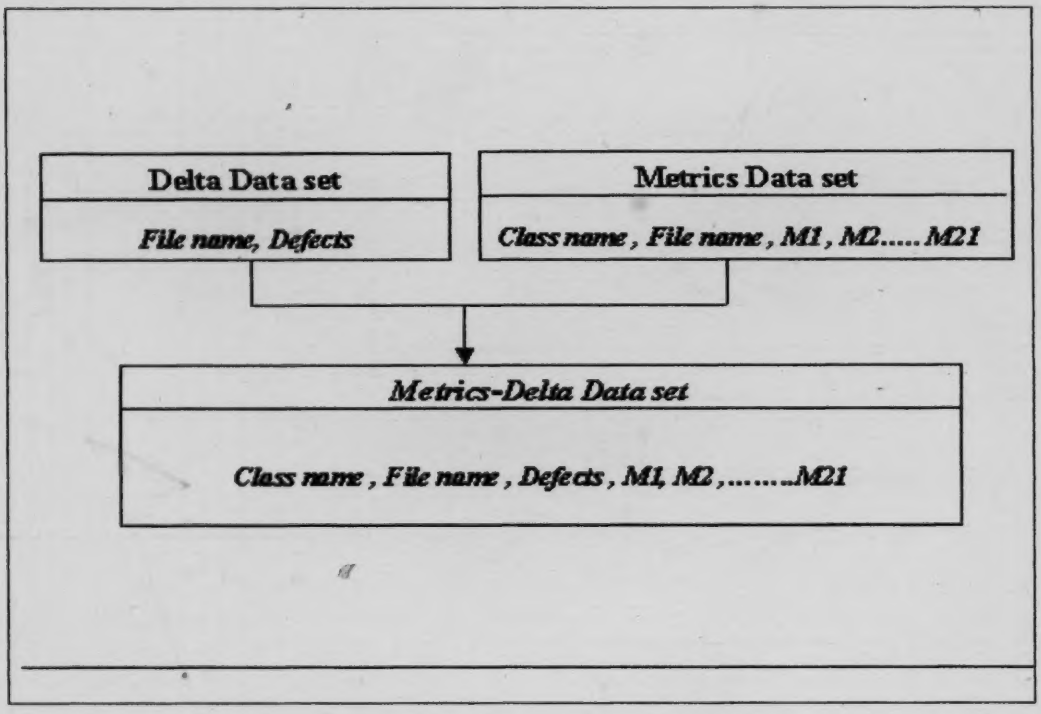
Delta Data set
File name, Defects
Metrics Data set
Class name , File name , M1 , M2..... M21
Metrics-Delta Data set
Class name , File name , Defects , M1, M2 ,.....M21

Figure 5.8: Joining Metrics-Delta Data Set

The creation Metrics-Delta data set was a two-step procedure.

1. Assignment of file defects to individual C++ classes
2. Joining the Delta and Metrics data sets to obtain Metrics-Delta data set.

4.1 Assignment of file defects to individual C++ Classes:

Assignment of defects from files to classes requires defects to be assigned for both the class definition and their respective method definitions. In order to achieve this task series of Perl programs were developed for assigning the defects to C++ classes according to their respective source lines of code (using .h files). Secondly, defects are assigned to classes for their method definitions according to the summation of source lines of code of all the methods that belong to an individual class (using .cpp files) and finally to get the total defects for a class we added defects for class definition and its respective methods definition.

The first program [Program 5.9] developed a text file from *Mozmetrics* report, which contains the file names, class names and Sloc (source lines of code per class excluding comment lines and white spaces).

```
c:/mozilla/xpcom/threads/nsEventQueueService.h
 nsEventQueueServiceImpl,26
```

Figure 5.9: File Names, Class Names and SLOC-1

The second program [Program 5.10] extracted all the classes that belong to a particular file along with their respective Sloc using Perl data structure hash of array and send the results to another text file [see Figure 5.10]. Since the defects counts provided by delta data set are at the level of files and we have to assign them to individual classes, it was required to retrieve all the classes that belong to a particular File and divide defect counts

according to the value of SLOC (Source lines of code). This division was required because 1) in Mozilla defect reporting system defects could only be reported against files not for individual classes. 2) There is no clean mapping from files to classes that exist in other OO projects [1]. 3) We needed an approximation to count delta per class. With no other information our options are i) uniform division of defects or ii) or a linear approximation. Given the frequently reported correlation between defects and LOC, we elected the linear approximation. Although this approximation breaks down for smaller modules [30] but we do not have universal non-linear model for this breakdown.

```
c:/mozilla/xpcom/threads/nsEventQueueService.h
nsEventQueueServiceImpl,26
```

Figure 5.10: File Names, Class Names and SLOC -2

Another Perl program [Program 5.11] was developed to add SLOC for all the classes in a file named as TLOC (Total lines of code for all the classes) and then each class's SLOC was divided by TLOC. This division was performed in order to get the percentage of each class in a file for the assignment of defects. We name this percentage as contribution factor.

```
c:/mozilla/xpcom/threads/nsEventQueueService.h
nsEventQueueServiceImpl,1.0
```

Figure 5.11: File Names, Class Names and TLOC

A class with higher value of SLOC will get higher portion of a defect since greater the lines of code per class the greater the chances of errors are there. The new text file [see Figure 5.11] contains file name, class name and contribution factor.

This next program [Program 5.12] was developed to assign the defect counts from files to classes by reading class names and their respective file names from [Figure 5.11] and then searching for that particular file in delta data set. Once the file name is matched the defects of file are multiplied with contribution factor of each class. The output of this final program is class name, file name and defect count for the class [Figure 5.12].

```
c:/mozilla/xpcom/threads/nsEventQueueService.h
 nsEventQueueServiceImpl,8
```

Figure 5.12: File Name ,Class Name and Delta

After assigning the defects to class definitions according to their source lines of code, the next step was to assign defects to all the methods that belong to the same class according to the source lines of code for each method.

The next program [5.13] created a text file [see Figure 5.13] from Mozmetrics report and provided method name, class name, file name and source lines of code per method [see Figure 5.13].

Using Perl data structure hash of arrays in program [Program 5.14], all the methods that belong to one file were extracted [see Figure 5.14].

File name =

Class name1::Method name1, SLOC

Class name2::Method name1, SLOC

```
c:mozilla/xpcom/threads/nsEventQueueService.cpp
=hash_enum_remove_queues,19

c:/mozilla/xpcom/threads/nsEventQueueService.cpp
=NS_IMPL_THREADSAFE_ISUPPORTS1,9

c:/mozilla/xpcom/threads/nsEventQueueService.cpp
=nsEventQueueServiceImpl::CreateMonitoredThreadEventQueue,5

c:/mozilla/xpcom/threads/nsEventQueueService.cpp
=nsEventQueueServiceImpl::CreateFromIThread,15

c:/mozilla/xpcom/threads/nsEventQueueService.cpp
=nsEventQueueServiceImpl::CreateEventQueue,23

c:/mozilla/xpcom/threads/nsEventQueueService.c
=nsEventQueueServiceImpl::DestroyThreadEventQueue,21

c:/mozilla/xpcom/threads/nsEventQueueService.cpp
=nsEventQueueServiceImpl::CreateFromPLEventQueue,15

c:/mozilla/xpcom/threads/nsEventQueueService.cpp
=nsEventQueueServiceImpl::PushThreadEventQueue,52

c:/mozilla/xpcom/threads/nsEventQueueService.cpp
=nsEventQueueServiceImpl::PopThreadEventQueue,35

c:/mozilla/xpcom/threads/nsEventQueueService.cpp
=nsEventQueueServiceImpl::GetThreadEventQueue,48

c:/mozilla/xpcom/threads/nsEventQueueService.cpp
=nsEventQueueServiceImpl::ResolveEventQueue,14

c:/mozilla/xpcom/threads/nsEventQueueService.cpp
=nsEventQueueServiceImpl::GetSpecialEventQueue,38
```

Figure 5.13: File Name, Class Name and Method Name-1

```
c:/mozilla/xpcom/threads/nsEventQueueService.cpp=
 hash_enum_remove_queues,19
 NS_IMPL_THREADSAFE_ISUPPORTS1,9
 nsEventQueueServiceImpl::CreateMonitoredThreadEventQueue,5
 nsEventQueueServiceImpl::CreateFromIThread,15
 nsEventQueueServiceImpl::CreateEventQueue,23
 nsEventQueueServiceImpl::DestroyThreadEventQueue,21
 nsEventQueueServiceImpl::CreateFromPLEventQueue,15
 nsEventQueueServiceImpl::PushThreadEventQueue,52
 nsEventQueueServiceImpl::PopThreadEventQueue,35
 nsEventQueueServiceImpl::GetThreadEventQueue,48
 nsEventQueueServiceImpl::ResolveEventQueue,14
 nsEventQueueServiceImpl::GetSpecialEventQueue,38
```

Figure 5.14: File Name, Class Name and Method Name-2

Another Perl program [Program 5.15] was written to add TSLOC for all the classes in a file and divide each class by TTSLOC where TTSLOC stands for total source lines of code for all classes in a file. This division is performed to get the percentage for assigning defects for each class in a file [see Figure 5.15].

```
c:/mozilla/xpcom/threads/nsEventQueueService.cpp
 nsEventQueueServiceImpl
0.90
```

Figure 5.15: File Name, Class Name and Contribution Factor

The program [Program 5.16] was then developed to assign defect count at the level of classes. For this assignment the delta data set is reopened, each file name and its respective defect count is read by the code and assigned to the class that belong to this particular file by multiplying defects with contribution factor [see Figure 5.16].

```
c:/mozilla/xpcom/threads/nsEventQueueService.cpp
  nsEventQueueServiceImpl
    9.9
```

Figure 5.16: File Name, Class Name and Class-Level Delta

```
C:/mozilla/xpcom/threads/nsEventQueueService.h,
nsEventQueueServiceImpl,  18
```

Figure 5.17: File Name, Class Name and Final Class-Level Delta

The last program [program 5.17] in this sequence, added defects assigned to class definitions and their respective methods [see Figure 5.17].

4.2 Joining the Delta and Metrics data sets to obtain Metrics-Delta data set

The assignment of defects (obtained from delta data set) to individual classes was performed in first step. The next and final step in the creation of Metrics-Delta data set was to connect metric values for individual classes with their respective defect counts [see Program 5.18]. To perform this connection file name and class name were used as the key and metric values were tied with their respective defect count. This final data set, which we name as Metrics-Delta data set [see Figure 5.18], is comprised of class name, filename, defects and metrics obtained for C++ classes from Krakatau [64].

```
c:/mozilla/xpcom/threads/nsEventQueueService.h,nsEventQueueServiceImpl,
0,2,7,0,5,0,1,26,2,0,5,0,0,0,1.6,2.2,2,100,5,16,86,11,18
```

Figure 5.18: Metrics-Delta Data Set

The number of points in the data set are 8349 where as the number of attributes are 25. In order to develop the Metrics-Delta Data Set there were total of eighteen Perl programs were developed. However after reviewing the codes , we recommend that Program 5.3 and Program 5.4 can be merged to reduce the I/O processing. Both of these programs are used to extract .h and .cpp files respectively from Fig 5.3 and the regular expressions can be merged into a single program. Same goes for programs 5.6 & 5.7. Both of these programs again does the same function on Fig 5.5 and could be merged easily.

The details of statistical characterization of the data set along with preliminary regression experiments are provided in the next chapter. Pearson's correlation between dependent and independent variables is computed as well as pair wise Pearson's correlation is also reported. The regression experiments are performed using linear regression, SMO regression and multiplayer perceptrons.

CHAPTER 6

1. Introduction

Software predictive modeling techniques predict the software quality by classifying the software modules into high-risk modules (modules which are likely to contain most of the faults) and low -risk modules (modules which are likely to contain less number of faults). The techniques for developing predictive models can be categorized in two main categories that are traditional classical statistical methods and data mining methods. Linear regression, support vector machines least square regression and robust regression all belong to classical statistical methods where as neural networks, fuzzy logic models, case based reasoning and decision trees belong to data mining group. The detailed comparison of techniques and their benefits, drawbacks and their suitability in particular cases is discussed in [44].

The statistical report of the data set reporting minimum, maximum, mean, kurtosis and skewness is also included in this chapter. Pearson's correlation between all attributes and delta is also computed. We have performed data mining using linear regression, SMO regression and multilayer perceptrons. The details of the experiments and results are given in this chapter.

2. Data Mining

The KDD process described by Fayyad [34] is composed of nine activities: learning the, application domain, creating a target data set, data cleaning and preprocessing, data reduction and projection, selection of data mining function, selection of data mining algorithm, data mining, and evaluation and use of discovered knowledge (see Figure 2.1).

The details of activities performed in each step for data mining on Mozilla source code and Bugzilla database are given as:

- ***Learning the application domain***: In order to understand the problem domain, we developed ER diagram of Bugzilla database (see chapter 4) by performing data reverse engineering. The analysis of the ER diagram helped in identifying the data that was required to transform into desired information and also helped in understanding Mozilla architecture.
- ***Creating a target data set***: Metrics-Delta data set was developed for data mining operation. The detail of development process is described in chapter 5. The data set contains 21 attributes and 8349 records.
- ***Data cleaning and preprocessing***: This step is usually required to remove noise from the data and substituting missing values of the desired variables. Since the Metrics-Delta data set was already in a very good shape, we were not required to perform this step explicitly.
- ***Data reduction and projection***: The original Metrics-Delta data set was composed of 25 attributes. Among these attributes, three attributes namely File name, Class name and CBO (coupling between objects) were eliminated. The File name and Class name were taken away because of they were not useful as inputs for data mining algorithms where as the value of CBO was zero through out the data set and could not contribute in providing any information. Our initial experiments then focus on creating regression models for the full data set.
- ***Choosing the data mining function***: The goal of conducting data mining for our research project is to predict software quality. We decided to develop predictive

model using Metrics-Delta data set. Since all of the attributes in the data set including the value of Delta were continuous variables, we decided to run regression experiments.

- ***Choosing the data mining algorithm***: We choose linear, regression, SMO regression and multilayer perceptrons for performing data mining. The details of these algorithms are discussed in chapter 2.
- ***Data Mining***: In order to develop the predictive model for performing data mining on Metrics-Delta Data set, the tool we have used is Weka. Weka developed at the University of Waikato in New Zealand and it stands for the Waikato Environment for Knowledge Analysis. The machine learning algorithms in Weka are written in Java. It provides several learning methods to develop predictive models via regression and classification. Since Metrics-Delta data set is continuous for all its attributes we performed regression experiments to develop predictive models using simple linear regression, multilayer perceptrons and SMO regression. The prediction error is measured using root mean square error. The results of the experiments are provided in this chapter.
- ***Interpretation and use of discovered knowledge***: The last chapter of the dissertation [see chapter 7] is devoted to the summary of results we obtained, their proposed usage and future directions for this particular data mining project.

2.1 Variables and Description

The variables in Metrics-Delta data set were

- File name-Source file
- Class name- C++ classes

- CBO-Coupling between objects
- CSA-Class size (Attributes)
- CSAO-Class size (Attributes and Operations)
- CSI-Class specialization Index
- CSO -Class size (Operations)
- DIT-Depth of Inheritance tree
- LCOM-Lack of cohesion methods
- LOC-Lines of code
- NAAC- Number of attributes added
- NAIC-Number of attributes inheritance
- NOAC- Number of Operations Added by a Class
- NOCC- Number of child classes
- NOIC- Number of operations inherited by children
- NpavgC-Average number of method parameters
- NOOC- Number of operations in a class
- Osavg- Average operation size
- PA- Private attribute usage
- PPPC-Percentage public/protected members
- RFC-Response for class
- SLOC-Source lines of code
- TLOC-Total lines of code
- WMC- Weighted methods in class
- Delta- Defect Count per class

The first three variables Class name, File name and CBO were taken away before running the experiments. Class name and File name were merely identifiers not predictive attributes were not useful for running regression experiments where as CBO has the uniform value of 0 in all instances and could not contribute to provide any useful information. In order to check the rational behind getting uniform 0 value of CBO , we picked up some random cases and checked manually. The manual observation was not in conjunction with the value provided by the tool and coupling does exist between the objects. There seemed to be some problem with the metric tool that failed to provide the correct results.

2.2 Statistical Results

The statistical results are given in table 6.1. The results showed that maximum number of defects reported for a class is 104 whereas minimum is 0. Such a high number of defects reported can easily analyze that this class needs to be considered for extensive testing. Another important aspect to be noted is the maximum value of DIT and NOCC that is 13 and 186 respectively. These metrics describes the complexity of the class. Those, classes that are so deep in hierarchy and are providing its methods to 186 sub classes should be taken into account for reducing complexity. The maximum value of LCOM is 4014 that is also indicating higher complexity in the source code. The value of Kurtosis (measure of peak-ness and flatness towards normal distribution) is also high for most of the attributes and for NOCC is reached up to 45.149. Kurtosis for any normal distribution

should be close to 0. Very high value of kurtosis for any variable represents too tall distribution and indicates the presence of problem.

The most important feature to be noted from the statistical report is the high skewness value for most of the attributes. Skewness describes the degree of asymmetry of a distribution around its mean. For all attributes except PPPC and NpavgC it is very high and has reached up to the value of 2865 for NOCC. Normal distributions must have skewness approximately 0. Highly skewed values indicate violation of normality.

Variable	Min	Max	Mean	Std. Dev.	Kurtosis	Skewness
CSA	0	108	2.580	6.500	6.050	55.067
CSAO	0	328	9.652	17.796	5.889	57.287
CSI	0	44	0.182	0.796	26.145	1411.309
CSO	0	274	7.068	13.151	6.106	59.884
DIT	0	13	0.504	1.117	3.314	1191.834
LCOM	0	4014	6.302	86.527	30.827	1191.834
LOC	0	10003	33.627	57.337	5.884	50.963
NAAC	0	108	2.589	6.555	6.090	55.492
NAIC	0	104	1.509	6.774	6.726	52.201
NOAC	0	226	5.917	10.982	6.844	76.558
NOCC	0	186	0.327	2.685	45.149	2865.077
NOIC	0	430	11.994	48.905	5.447	31.173
NOOC	0	89	1.161	5.129	8.443	91.378
NPavgC	0	10	0.762	0.961	1.859	6.426
OSavg	0	33	1.530	1.522	5.296	44.644
PA	0	279	2.391	10.997	9.515	131.698
PPPC	0	150	87.740	26.644	-2.193	3.748
RFC	0	675	10.247	26.842	9.735	153.224
SLOC	1	795	21.675	31.764	6.511	80.306
TLOC	0	10280	112.026	334.249	10.633	191.539
WMC	0	1472	15.746	48.945	11.432	214.463
Delta	0	104	0.140	1.49	44.139	2856.022

Table 6.1: Statistical Results of Metrics-Delta Data Set

Table 6.2 reports the Pearson's correlation coefficient between each metric value and the Delta. Important point to be noted here is the presence of very weak correlation between metrics values and defect counts (Delta).

Metric	Correlation to Delta
CSA	0.071934
CSAO	0.112962
CSI	0.504449
CSO	0.09769
DIT	0.107877
LCOM	0.047869
LOC	0.103314
NAAC	0.175611
NAIC	0.048786
NOAC	0.072955
NOCC	0.049905
NOIC	0.020787
NOOC	0.098526
NPavgC	0.044162
OSavg	0.036324
PA	0.068462
PPPC	-0.04419
RFC	0.118019
SLOC	0.129722
TLOC	0.106383
WMC	0.20005

Table 6.2: Correlation of Metrics to Delta

	A	B	C	D	E	F	G	H	I	J	K	L	M	NA	O	P	Q	R	S	T	U	V
A	1	0.80	0.01	0.59	0.05	0.39	0.49	0.98	0.02	0.57	-0.0	0.05	0.27	0.19	0.26	0.51	-0.1	0.49	0.61	0.46	0.45	0.07
B		1	0.11	0.95	0.15	0.47	0.78	0.79	0.10	0.89	0.89	0.17	0.53	0.30	0.28	0.54	-0.0	0.82	0.85	0.69	0.70	0.11
C			1	0.13	0.67	0.02	0.08	0.10	0.31	-0.0	0.04	0.56	0.35	0.21	0.11	0.07	-0.0	0.15	0.11	0.10	0.18	0.50
D				1	0.17	0.44	0.82	0.58	0.12	0.92	0.04	0.21	0.58	0.31	0.24	0.47	-0.0	0.87	0.84	0.71	0.73	0.09
E					1	0.03	0.13	0.07	0.45	0.05	0.02	0.77	0.35	0.27	0.15	0.09	-0.0	0.20	0.16	0.16	0.16	0.10
F						1	0.27	0.39	0.01	0.38	0.00	0.04	0.30	0.04	0.07	0.35	-0.0	0.48	0.34	0.38	0.39	0.04
G							1	0.48	0.06	0.79	0.04	0.16	0.41	0.36	0.24	0.40	-0.0	0.73	0.88	0.66	0.61	0.10
H								1	0.03	0.57	0.00	0.05	0.27	0.19	0.26	0.50	-0.1	0.49	0.60	0.45	0.46	0.17
I									1	0.03	0.01	0.53	0.23	0.12	0.04	0.04	0.03	0.11	0.08	0.06	0.07	0.04
J										1	0.03	0.07	0.23	0.27	0.23	0.37	-0.0	0.79	0.78	0.65	0.67	0.07
K											1	0.03	0.04	0.04	-0.0	0.02	-0.0	0.04	0.03	0.03	0.04	0.04
L												1	0.38	0.19	0.42	0.10	0.04	0.23	0.19	0.18	0.16	0.02
M													1	0.21	0.14	0.42	0.01	0.53	0.48	0.44	0.42	0.09
N														1	0.28	0.14	-0.0	0.28	0.36	0.29	0.27	0.04
O															1	0.28	-0.0	0.31	0.32	0.50	0.51	0.03
P																1	0.07	0.53	0.45	0.56	0.54	0.06
Q																	1	-0.0	-0.0	-0.0	-0.0	-0.0
R																		1	0.78	0.87	0.8	0.11
S																			1	0.73	0.6	0.12
T																				1	0.94	0.10
U																					1	0.20
V																						1

Table 6.3 (page 105). Along with the Pearson's correlation between each metric and Delta, we have also computed pair wise Pearson's correlation and are reported in Table 6.3. These results indicate the presence of complex correlation that is a combination of strong and weak correlation between attributes. Due to the size of table we have labeled the variables as alphabetic characters. The description of labels is given in table 6.4

Variable	Label
CSA	A
CSAO	B
CSI	C
CSO	D
DIT	E
LCOM	F
LOC	G
NAAC	H
NAIC	I
NOAC	J
NOCC	K
NOIC	L
NOOC	M
NPavgC	N
OSavg	O
PA	P
PPPC	Q
RFC	R
SLOC	S
TLOC	T
WMC	U
Delta	V

Table 6.4: Variable Abbreviations

Examining the results of Table 6.2 , it is observed that all three metrics associated with class size that are LOC, SLOC & TLOC posses very small correlation with Delta. So the impact of class size on defect density should be strictly minor. Based on correlation results , we can conclude that there should be no impact of class size on the KDD process.

2.3 Experiments

The experiments were performed using a 10-fold cross validation design. In 10-fold cross-validation data is divided into 10 disjoint partitions. In each turn separate 9 partitions are used for training and the remaining 1 partition is used for testing. Model performance was evaluated by calculating root mean square of error values. The root mean square (RMS) value is given by

$$x_{rms} = \sqrt{1/N \sum_{i=1}^{N} x_i^2}$$

Where N is the total number of profiles. The smaller is the value of RMS, the better the performance of model is. Since obtaining lowest error measures for a given data set is a trial and error procedure in case of smo regression and mulitlayer perceptrons. The parameter for tuning smo regression in order to minimize the error rate is *C*, which is a complexity parameter. This parameter pays the penalty to obtain the minimum value of error measure. We ran different experiments ranging the value of *C* from 1-100 and the best results were obtained where *C*=5.

Again for multilayer perceptrons different combination of neurons per layer were tried. Number of hidden layers varied from 1 to 3. Number of neurons tried in different combinations between layers ranged between 1-100. The best results for this data set

were obtained for network with one hidden layer and one neuron in that layer. The root mean square error values for these experiments are given in table 6.4.

The experimental results showed that linear regression and mulitilayer perceptrons provided almost same results. SMO regression was a bit better as compared to linear regression and multiplayer perceptrons.

```
DELTA =

      0.0071 * CSA +
      1.5314 * CSI +
     -0.0131 * CSO +
     -0.3179 * DIT +
      0.0015 * LOC +
      0.0108 * NAIC +
     -0.0064 * NOAC +
      0.0131 * NOCC +
     -0.0072 * NOIC +
     -0.0251 * NOOC +
     -0.0651 * NPavgC +
     -0.0829 * OSavg +
     -0.002  * RFC +
      0.0089 * SLOC +
     -0.0013 * TLOC +
      0.0143 * WMC +
      0.1035
```

Figure 6.1: Linear Regression Model

```
DELTA =
        -0.4284 * (normalized) CSA
 +       0.1252 * (normalized) CSAO
 +       0.0024 * (normalized) CSI
 +      -0.1523 * (normalized) CSO
 +       0.0006 * (normalized) DIT
 +      -0.0008 * (normalized) LCOM
 +      -0.0009 * (normalized) LOC
 +       0.3876 * (normalized) NAAC
 +       0.0002 * (normalized) NAIC
 +       0.0389 * (normalized) NOAC
 +      -0.0018 * (normalized) NOCC
 +      -0.001  * (normalized) NOIC
 +       0.0162 * (normalized) NOOC
 +      -0.0002 * (normalized) NPavgC
 +      -0.0002 * (normalized) OSavg
 +      -0.0001 * (normalized) PA
 +      -0.0004 * (normalized) PPPC
 +       0.0009 * (normalized) RFC
 +       0.0015 * (normalized) SLOC
 +      -0.0007 * (normalized) TLOC
 +      -0.0005 * (normalized) WMC
 +       0.0018
```

Figure 6.2: SMO Regression Model

For linear regression the value of weight assigned to CSI was high which dropped very low for all other attributes. The assignment of weights for SMO regression didn't showed any drastic change among all variables

	Average	Standard Deviation
Linear Regression	**1.3459**	**0.8572**
Multilayer Perceptrons	**1.3685**	**1.0870**
SMO Regression	**1.2001**	**0.9043**

Table 6.5: Regression Results

The prediction results were not very good, however, the overall quality of the models was reasonable. This was due to the fact that data set was highly skewed in nature. Skewness causes the distortion in machine learning algorithm and the model fails to optimize global

performance index. There are two main approaches, which deals with this problem. One is data re-sampling and other is the assignment of penalty for the learner. In the area of software metrics Khoshgoftar.,[61] has proposed to use differing misclassification penalties in a decision tree algorithm. Dick., [10] has proposed the use of re-sampling using SMOTE and then performing data mining using decision tree learner. To achieve the better prediction results from highly skewed data set we would recommend the classification of the data set. Discretization reduces the variance of dependent variable, making it easier for learning algorithm.

The summary of achievements for conducting this research and their proposed usage along with future directions are provided in chapter 7.

CHAPTER 7

1. Summary

The objective of this research was an attempt towards improving software quality. We have performed the KDD process for software quality prediction. The description of our effort in mining software quality data from a large-scale open-source software system and their proposed usage is given below.

Development of ER diagram of Bugzilla database: By performing data reverse engineering on Bugzilla database source code, we developed the complex ER diagram composed of 34 entities and 150 attributes. Since Bugzilla is a defect tracking system of Mozilla web browser and like most of the open source products doesn't provide sufficient amount of conceptual documentation. This diagram will help in understanding Bugzilla database for those who are interested in conducting any research on Mozilla. Currently, its been used for the laboratory manual of CMPE 440 at the department of Electrical and Computer Engineering, University of Alberta, Canada.

Development of Metrics-Delta Data set: Based on Mozilla source code and Bugzilla database the data set is developed. This data set contains twenty-two attributes among which twenty-one are class-level metric values for C++ classes in Mozilla source code and the final attribute Delta is the number of defects reported for each class. The data set contains eight thousand, three hundred and forty nine points. Although there are twenty-two metric attributes but for the discussion here, we are only considering defect counts and classical CK metrics. The statistical report of the data set described the defect density and complexity of classes in Mozilla source code. The maximum defects reported for a class is one hundred and four. A class with such a high value of Delta needs to be

considered for careful testing. The maximum values of CK metrics clearly indicate presence of very complex design. For example, the maximum values of DIT and NOCC are thirteen and one hundred and eighty-six respectively. Both of these attributes represent complexity of classes. The behavior of a class with high DIT value becomes unpredictable and it is recommended that the design of class needs to reviewed where as the higher value of NOCC is suggesting that the methods of the class are subjected to extra testing effort. The value of WMC is one thousand, four hundred and seventy-two. A class with high WMC value needs extra maintenance effort and its potential reuse is decreased. The maximum value of LCOM is four thousand and fourteen. LCOM is a measure of lack of cohesiveness and good object-oriented programming practice suggests that classes should be cohesive in nature. A very high value of LCOM is indicating that this class requires a breakdown. And finally, if we observe the maximum value of RFC that is six hundred and seventy-five, there is a very clear indication of debugging and extra testing effort required for this class.

Development of Predictive Models: Predictive models using linear regression, SMO regression and multilayer perceptrons are built. The results obtained from SMO regression provided slightly better predictive model as compared to linear regression and multilayer perceptrons. The overall quality of predictive models was reasonable however they did not provided desirable lower error measures.

2. Future Directions

1. The values in the Metrics-Delta data set are continuous and could be used for regression experiments only. Discretization of Delta value will allow performing classification experiments as well. The classification experiments can divide the

classes into high and low-risk categories. Identification of high-risk modules would be very helpful for approximating effort required to perform testing and can help software managers in project management. Providing mining results in that format makes their job easier.

2. Along with project managers it would also help software testers. Testers can use this information to prioritize their testing and focus their efforts to make the testing process more efficient and the resulting software is more dependable or, indicate when re-factoring is required.
3. Executing the codes developed for creation of data set on another instance of Bugzilla database (which we already got from Bugzilla organization) can compare the quality of Mozilla web browser at different points in time. It will demonstrate the evolution of software quality in a long-lived open-source project. This could contribute to our understanding of the evolution of software quality in a long-lived open-source project.
4. Mining past bug history of a software project can be used as a guide in determining what types of bugs should be expected in current snapshot and also can help in recommending which of a group of bug reports are more likely to be true. This would address the issue of software quality assurance by focusing developer's attention on those modules, which are likely to contain defects.
5. Open-Source software projects may not always keep good historical records, even in their bug-tracking databases. Maintaining bug history by identifying defect density at different points in time may also help in improving software engineering practice .

6. One of the problems associated with open source product is presence of large number of duplicate and redundant bugs posted by open source community. A study on Apache [74], suggested that less than two percent of reported bugs were actually those, which requires patches to get fixed. We also support that by our observation. Out of 190722 reported bugs only 4665 bugs were fixed using patches.

REFERENCES

1. Akif G. Koru and Liu H., "An Investigation of the Effect of Module Size on Defect Prediction Using Static Measures", PROMISE - Predictive Models in Software Engineering Workshop, ICSE 2005, May 15th 2005, Saint Louis, Missouri, US.

2. Alonso, O.; Premkumar, T. D.; Gertz, M., "Database Techniques for the analysis and exploration of software repositories", International Workshop on Mining Software Repositories. 25th May 2004, Edinburgh, Scotland, UK.

3. Bagui, S.; Earn, R., " Database Design Using Entity Relationship Diagrams", Auerbach Publications , 2003.

4. Baisch. E; Bleile. T.; Belschner.R "A neural fuzzy system to evaluate software development productivity", IEEE International Conference on Systems, Man, and Cybernetics, Intelligent Systems for the 21st century , Vancover, BC, Canada, October 22-25, 1995.

5. Baische. E; Liedtke. T, "Comparison of conventional approaches and soft computing approaches for software quality prediction", IEEE International Conference on Systems, Man and Cybernetics, Orlando, Florida, U.S.A. ,October 12-15 , 1997.

6. Bechtold, R ., "Essentials of software project management"., August, 1999, Management Concepts, Incorporated.

7. Berry, D.M. ; Lawrence, B. Software., "Requirements Engineering". IEEE vol-15, no 2, March -April 1998, pp26-29

8. Bleile. T.; Baisch. E; Belschner. R, " Neural fuzzy simulation to gain expert knowledge for the improvement of software development productivity", Proceedings of the 1995 Summer Computer Simulation conference (27th) Ottawa, On, Canada, July 24-25 1995.

9. Blaha, M., "On Reverse engineering of vendor databases". In working conference on Reverse Engineering (WCRE-1998), Honululu, Hawaii, USA, pages 183-190. IEEE computer society press, October 1998.

10. http://bloof.sourceforge.net

11. Briand, C.V. ; Basili. R; Thomas, W. M., "A pattern recognition approach for software data analysis, IEEE Transactions on Software Engineering, vol.18 , pp 931-942, 1992.

12. Briand. L; Thomas, W.M ; Hetmanski, C.J. " Modeling and managing risk early in Software Development", Proceedings of 15th Conference on Software Engineering , 1993, pp. 55-65.

13. Brown, J. R., DeSalvio A. J., Heine, D. E. and Purdy, J. G. "Automatic Software Quality Assurance," in Program Test Methods, W. C. Hetzel (Ed.), Prentice-Hall, Englewood Cliffs, N.J., 1973, pp. 181-203.

14. Brooks, Jr., F. P., 1987 , "No Silver Bullet--Essence and Accidents of Software Engineering," The Mythical Man Month, Essays on Software Engineering, Anniversary Edition .Published by Addison Wesley 1995.

15. Boehm, B, W. ,"A Spiral model of software development and enhancement", IEEE Computer, vol.21 no 5,May 1988, pp 61-72.

16. http://www.bugzilla.org/

17. Buckle,G., "Static Analysis of Safety Critical Software", Proceedings of the sixth safety critical system symposium ,1998.

18. Christanini, N.; Shawe-Taylor, J., "An introduction to support vector machines and other Kernal based learning methods", Cambridge University Press, 2000.

19. Chaar, J.K.; Halliday, M.J.; Bhandari, I.S.; Chillarege, R., "In-process evaluation for software inspection and test ", IEEE Transactions on Software Engineering, vol. 19 , no. 11 , Nov. 1993 ,pp. 1055 – 1070

20. Chidamber, S., R.; Kemerer, F., C., "A metrics suite for object oriented design " IEEE transactions on Software Engineering, , June 1994 , vol. 6, no.20.

21. Clark, D "An Introduction to Object-Oriented Programming with Visual Basic .NET ", Apress, 2002.

22. Curtis, B.; Kellner, M., I; Jim, O., " Process Modeling ", Communications of the ACM, Sep 1992, vol. 35, no. 9.

23. Daniel M. G., "Mining CVC repositories, the softChange experience" MSR 2004, International Workshop on Mining Software Repositories. 25th May 2004, Edinburgh, Scotland, UK.

24. Danielle, B.; March, S., "An approach for analyzing the information content for existing database", Database 1989, pages 1-8.

25. Dayani-Fard, H. and I . Jurisica, " Reverse Engineering by mining dynamic repoitores", Proceedings of the Fifth working conference on the reverse Engineering , October 1998, Honululu, Hawai , pp. 174-182.

26. De Almeida, M.A ; Lounis. H ; Melo, W. L. , "An investigation on the use of machine learning models for estimating software correctibility", International journal of software engineering and Knowledge engineering, vol. 9,.no.5, pp. 565-593, October 1999.

27. De Almeida, M. A; Lounis. H; Melo, W.L, "An investigation on the use of machine learning models for estimating software correctibility ". International Journal of Software Engineering and Knowledge Engineering, vol. 9, no.5, pp. 565-593, 1999.

28. Deitel., "C++ how to program ", Prentice Hall, Fourth Edition .

29. Draheim, D. ; Pekaei , L., " Process-Centric analytical processing of version control data", In IWPSE , Helsink, Finland, September 2003. IEEE press.

30. Dick. S., "Computational Intelligence in Software Quality"(PHD Dissertation 2002)

31. Ebert. C "Fuzzy classification for software criticality analysis", Expert Systems with applications, vol .11, no. 3, pp.3223-342, 1996.

32. Elmasri, R.; Navathe, S.,B., " Fundamentals of Database Systems "Addison Wesley, Third Addition.

33. Fayyad; Shapiro.P; Smyth "From Data Mining to Knowledge Discovery: An Overview", Advances in Knowledge Discovery and Data Mining, AAAI Press / The MIT Press, Menlo Park, CA, pp.1-34, 1996.

34. Fayyad, U. M.; Piatetsky-Shapiro, G.; Smyth, P., " The KDD process for extracting useful knowledge from volumes of data", Communication of the ACM, vol. 39, no. 11, 1996, pp. 27-34.

35. Fayyad, U.; Haussler, D.; Stolorz, P.," Mining Scientific Data", Communications of the ACM, vol. 39, no. 11, Nov 1996.

36. Fairley, R. E , " Static analysis and dynamic testing of computer software(Tutorial)".

37. France, R.B.; Kim, D.-K.; Sudipto Ghosh; Song, E., "A UML-based pattern specification technique", IEEE Transactions on Software Engineering, vol. 30 , no. 3 , March 2004, pp. 193 –196.

38. Friedman, M.A.; Voas, J., M., "Software Assessment: Reliability, Safety, Testability" , New: John Wiley & Sons, Inc., 1995.

39. Fischer, M.; Pinzger, M.; Gall, H., "Populating a release history database from version control and bug tracking systems", Proceedings of the International Conference on Software maintenance, Amesterdam , Netherlands, September 2003.

40. Gamma, E.; Helm, R.; Johnson, R; and Velissides, J.," Design Patterns: Abstraction and Reuse of Object-oriented design ", January 2002, Software Pioneers.

41. Gacek, C.; Arief, B.; "The many meanings of open source", IEEE Computing Society, Jan-Feb 2004.

42. German, D.; Mockus, A., " Automating the measurement of open source projects", In proceedings of ICSE'03 . Workshop on open source software engineering, Portland, Oregon, May 2003

43. Gray, A.R; MacDonell, Stephen G. (1996): "A comparison of techniques for developing predictive models of software metrics", Information and Software technology vol. 39, 425-437, 1997.

44. Gray. A.R "A simulation-based comparison of empirical modeling techniques for software metric models of development effort". Proceedings of 6th International Conference on Neural Information Processing: Perth, Australia, 16-20 November 1999 .

45. Halstead , M ., Elements of Software Science, New York: Elsevier, 1977.

46. Halpin, T., "Information Modeling and Relational Databases: From Conceptual Analysis to Logical Design" , Morgan Kaufmann Publishers,2001

47. Hofmann, H.F.; Lehner, F.;Software , "Requirements Engineering as a success factor in software projects", IEEE ,vol 18 , no 4 , July-Aug. 2001 ,pp:58 – 66

48. Hofmann, M.; Tierney, B., "The involvement of Human Resources in large scale data mining projects", Proceedings of the 1st international symposium on Information and communication technologies ,Dublin, Ireland, pp. 103 – 109, 2003.

49. Hogshead, D., K.; Adarsh K. A. ," A methodology for translating a convention al file system into Entity-Relationship model", Proceedings of the fourth International conference on Entity relationship approach, 1987, pp.148-159.

50. Hogshead, D., K.; Adarsh K. A. "Converting a relational database model into entity-relationship model", Proceedings of the sixth International conference on Entity relationship approach, 1989.

51. IEEE, Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990), Software Engineering Technical Committee, IEEE Computer Society, Los Alamitos, CA.

52. IEEE standard for software maintenance 1219-1993.

53. IEEE standard for Software test documentation 829-1998.

54. Jang, S.R ; Sun, C.T; Mizutani.E , "Neuro-Fuzzy and Soft Computing , A computational approach to learning and Machine Intelligence".

55. Jean-Luc, H., "Database Reverse engineering: Models, techniques and strategies", Proceedings of the tenth International conference on Entity relationship approach, 1991.

56. Jeffery, S. N., "Mission-Critical Development with open source software: Lessons learned" , IEEE software Jan / Feb 2004.

57. Johns, T.C., "Measuring Programming Quality and Productivity", IBM System Journal, vol. 17, no. 1, 1978

58. Jorgenson, M., "Experience with the accuracy of software maintenance task effort prediction models", IEEE TSE vol. 21, no. 8, pp. 674-68, 1995.

59. Karim, K.H., " Exploring Data Mining Implementation ", Communications of the ACM , vol. 44, no.7 , pp 87-93, July 2001.

60. Khoshgoftar, T.M; Allen, E.B; Jones, W.D, " Data mining for predictors of software Quality", International Journal of software Engineering and Knowledge Engineering vol. 9, no. 5,1999.

61. Khoshgoftar, T.M; Allen, E.B; Jones, W.D, Hudepohl P. John, "Classification-Tree Models of Software-Quality Over Multiple Releases", IEEE Transactions on Reliability, vol. 49, no. 1, March 2000

62. Krakatu professional for C++.

63. Krishnamurthy, S., " Cave or Community? An Empirical Examination of 100 mature open source projects", First Monday, vol. 7, no. 6 , June 2002.

64. Koch. S., "Free/Open Source Software Development ", Idea Group Publishing, 2005

65. Lakhani, K., R.; Wolf, R., " Why hackers do what they do: Understanding motivation and effort in free/open source software projects", Perspectives on Free and Open Source Software , 2005.

66. Lattemann, C.; Stieglitz, S., "Framework for Governance in open source communities " , Proceedings of the 38th Hawai International Conference on System Sciences , 2005.

67. Lee, S.C .:Lee, E.T , "Fuzzy sets and neural networks", Journal of Cybernetics, vol. 4 no.2, pp. 83-103,1974.

68. Len, B.; Clements, P.; Kazman, R., "Software Architecture in Practice (2nd edition), ; Addison-Wesley 2003.

69. Leveson, N.,G., "Stretching the limits of Software engineering" , Communications of the ACM, Feb 1997, vol. 40, no. 2.

70. Li , Y.E , "Applications artificial neural networks and their business applications" , Information and Management, vol. 27, pp. 303-313, 1994.

71. Mary, S.; Garlan, D, "Software Architecture: Perspectives on an emerging discipline ", Upper Saddle River, NJ Prentice Hall , 1996.

72. Marnie, L. H., " Software Testing Fundamentals: Methods and Metrics". John Willey & Sons.2003

73. Majardomo- A Bugzilla discussion group.

74. Mehmud, K. " Data Mining: Concepts, Models, Methods, and Algorithms" . John Wiley & Sons, 2003.

75. Ming-Syan , C.; Han, J; Yu, P.S.," Data mining: an overview from a database perspective", IEEE Transactions on Knowledge and Data Engineering, vol. 8 , no. 6 , Dec 1996, pp .866 – 883

76. Mertoguno, J.S; Paul.R; Bourbakis, N.G; Ramamoorthy, C.V "A Neuro-Expert system for the prediction of software metrics". Engineering Application.Artif . Intel, vol . 9, no. 2, pp. 153-161, 1996.

77. Mendonca, M.G; Basili.V.R; Bhandari ,I.S; Dawson.J, "An approach for improving existing measurement frameworks". IBM Systems Journal, vol. 37, no .4. 1998.

78. McLellan.S; Al Roesler; Fei.Z, Chandran.S; and Spinuzzi.C, "Experience using web-based shotgun measures for large –system characterization and improvement", IEEE Transactions on Software Engineering, vol. 24, no.4 ,April 1998.

79. Mendonca, M.G; Basili, V.R., " Validation of an approach for improving existing measurement frameworks", IEEE Transactions on Software Engineering, vol.26, no. 6, June 2000.

80. McCabe, T.J, " A complexity Measure", IEEE Transactions on Software Engineering, vol. 2 no.4, pp . 308-320.,December 1976.

81. McLaughlin, L .,"Automated bug tracking system : the promise and pitfalls", IEEE Software, vol. 21, no. 1, pp. 100- 103, Jan-Feb 2004.

82. McConell, S., "Software Project survival guide", Microsoft Press 1998.

83. Ministry of Defense "Requirements for safety related software in Defense Equipment." Part 1, Issue 2, Defense standard 00-55, August 1st 1997.

84. Microsoft Visual SourceSafe, http://www.microsoft .com/ssafe/.

85. Mockus, A.; Roy, T .; Herbsleb, J.,D., "Two case studies of Open source software development: Apache and Mozilla", ACM Transactions on software engineering methodology , vol. 11, no. 3, July 2002, pp. 309-346.

86. http://www.mozilla.org/

87. http://www.mozilla.org/roadmap.html

88. http://www.mozilla.org/projects/seamonkey/

89. http://www.Mozilla.org/docs/.

90. Muller, H., A.; Jahnke, J.H.; Smith, D.; Storey, M.; Tilley, S.; Wong, K., "Reverse Engineering: A Roadmap", Proceedings of the Conference on The Future of Software Engineering, May, 2000

91. Nilson, E., G., "The translation of COBOL data structure to an Entity relationship type conceptual schema", Proceedings of the Fourth International conference on Entity relationship approach, 1987, pp. 170-177.

92. Open Source Initiative. The Open Source definition, 1997. http://www.opensource.org/osd.html.

93. O'Reilly, T. , " lessons from open-source software development", Communications of the ACM, 1999, vol. 42 , no. 4 , pp. 32-37,

94. Ostrand . J.; Weyuker, E., J., Bell, R.,M., "Where the Bugs are", Proceedings of International Symposium on Software Testing and Analysis (ISSTA2004) ,Boston, MA, July 2004.

95. http://www.openoffice.org/licenses/gpl_license.html

96. P. Aiken., " Data Reverse Engineering: Slaying the Legacy Dragon", McGraw-Hill, 1995.

97. Patton, Ron, "Software Testing" , Indianapolis, Indiana: Sams publishing, 2001, pp1-124, 281-316

98. Parnas, D.L.; Lawford, M., "The role of inspection in software quality assurance", IEEE Transactions on Software Engineering , vol. 29 , no. 8 , Aug.2003

99. Parnas, D.,L.; John. A.; Kwan, P., "Evaluation of Safety-Critical Software", Communications of the ACM, June 1990, vol. 33, no.6.

100. Paul, R.A; Tosiyasu L. Kunii, Yoshihisa Shinagawa, and Muhammad F.Khan "Software metrics knowledge and databases for project management", IEEE Transactions of Knowledge and Data Engineering, vol.11., no. 1,1999.

101. Paulson, J.,W.; Succi, G.; Eberlein, A., "An Empirical study of Open-Source and Closed-Source software products", IEEE transactions on Software Engineering, vol. 30, no. 4, April 2004.

102. Peters J.F.; Pedrycz W., "Software Engineering :An Engineering approach" , New York : John Wiley & Sons,2000.

103. P. Fleeger, Shari Lawrence: "Software Engineering, Theory and Practice." Second Addition. Prentice Hall.

104. Pham.H , "Software Reliability", New York 2000.

105. Porter.A . and Selby.R, "Empirically-guided software development using metric-base classification trees", IEEE Software vol. 7, no. 2 , pp.46-54, 1990.

106. Pressman, R., S., "Software engineering a practitioner's approach "McGraw-Hill, 1996, Fourth Edition

107. Premerlani, W.,J.; Blaha, M.,R., "An approach for reverse engineering of Relational Databases", Communications of the ACM, vol. 35, no. 9 September 1992

108. Platt, J., " Sequential Minimal Optimization: A fast training Algorithm for training support vector machines ",Advances in Kernel Methods- Support Vector Learning, pp.185-208, Cambridge, M.A, MIT Press.

109. Rational ClearCase, http:// www.rational.com/products/clearcase/.

110. Raymond, E., " The Cathedral and the Bazaar", Linux Kongress, May 1997.

111. "Report on the NATO Software Engineering Conference, Garmisch, 1968" in Software Engineering Concepts and Techniques, P. Naur and B. R. Randell (eds.), Petrocelli/Charter, 1976.

112. Robert, B., "Transformational implementation: an example", IEEE Transactions on Software Engineering, SE, vol.7, no.1, p. 3-14, Jan. 1981.

113. Ruffin, M.; Ebert , C., "Using open source software in product development: A primer ", IEEE Software Jan/Feb 2004.

114. Ruff, J., CMPE 440 , Software Systems Design Project Lab manual, University of Alberta , Canada.

115. Smola, J.,A; Scholkopf, B., "A tutorial on support vector regression " Kluwer Academic Publishers 2004.

116. Schenker, D.F; Khoshgoftar, T.M, "The application of fuzzy enhancement case-based reasoning for identifying fault prone models", Proceedings of Third IEEE

117. International High-Assurance Systems Engineering Symposium, pp. 90-97, Washington, DC, USA, 13-14 Nov 1998 .

118. Shaver, M.; "Inside the Lizard A Look at mozilla technology and architecture", .www.mozilla.org/docs/ora-oss2000/ arch-overview/moz-arch-overview.pdf

119. Shin.M ; Goel, A.L, "Knowledge discovery and validation in software metrics databases", Part of the SPIE Conference on Data mining and Knowledge Discovery Theory, Tools and Technology, 1991.

120. Shirabad, J.,S.; Timothly C. L.; Matwin, C.; "Mining the software repository of a legacy telephone system", MSR 2004, International Workshop on Mining Software Repositories. 25th May 2004, Edinburgh, Scotland, UK.

121. http://sourcechange.sourceforge.net

122. Spinellis, D.; Szyperski, C., " How is open source affecting software development?", IEEE Software published by IEEE computer society. Jan-Feb 2004.

123. Stamelos, I.; Angelis, L.; Oikonomou, A.; Bleris, G.,L., "Code quality analysis in open source software development " Information Systems Journal, 2000, vol. 12, pp. 43-60.

124. Thomas J.O.; Weyuker, E.,J., "A tool for mining Defect Tracking systems to predict fault-prone files" ", International Workshop on Mining Software Repositories. 25th May 2004, Edinburgh, Scotland, UK.

125. Vapnik, V., "Statistical Learning Theory" New York: Wiley, 1998.

126. Van. F; Demeyer, S., "Mining version control systems for frequently applied changes", MSR 2004, International Workshop on Mining Software Repositories. 25th May 2004, Edinburgh, Scotland, UK.

127. Wang, J., "Data Mining: Opportunities and Challenges ", Idea Group Publishing 2003

128. Whittaker, J.A ; Jorgensen. A, " Why Software Fails", Software Engineering Notes, vol. 24 no. 4,July 1999, pp. 81-83.

129. Waters, R.C., "The programmer's apprentice: A session with KBEmacs", IEEE Transactions on Software Engineering , vol.11 no.11, ,pp.1296-1320, November 1985.

130. Wittig. G, "Estimating Software Development effort with connectionist models Working Paper Series ", Monash University, 1995.

131. Williams, C.,C.; Hollingsworth, J. K., "Bug driven Bug finders", International Workshop on Mining Software Repositories. 25th May 2004, Edinburgh, Scotland, UK.

132. XU.Z; Khoshgoftar, T.M; Ellen, E.B, " Prediction of software faults using fuzzy linear regression modeling", Proceedings of the Fifth IEEE Symposium on high assurance, Albuquerque, New Mexico, November 15-17 2000.

133. Xu N., " An Exploratory study of Open Source Software based on Public Project Archives", Thesis, the John Molson School of Business, Concordia University, Canada, 2003

134. Yuan.X; Koshgoftar, T.M; Allen, E.B, "An application of fuzzy clustering to software quality prediction", 3rd IEEE Symposium on Application –Specific Software Engineering technology, Richardson , Texas, March 24-25, 2000.

135. Yang, H; Ward, M., " Successful Evaluation of Software Systems" Artech House 2003

136. Zimmermann, T.; Weibgerber, P., "Preprocessing CVS data for Fine-Grained Analysis", International Workshop on Mining Software Repositories. 25th May 2004, Edinburgh, Scotland, UK.

APPENDIX A

Program 5.1

```
#!/usr/bin/perl
open (OUTF, ">>C:/perl/bin/mybugs2.txt")   or die $!;
open (INF, "C:/perl/bin/mybugs1.txt")   or die $!;
@data1=<INF>;
foreach $data1(@data1)
   {
       @data1= grep(/^RCS file:/,@data1);
    }
print OUTF @data1;
close(INF);
```

Program 5.2

```
open (INF, "C:/perl/bin/mybugs2.txt")   or die $!;
while ($line= <INF>)
 {
     $line =~ s/RCS file\://;
     $line =~ s/\/cvsroot\/mozilla//;
     $line=~s/\\//;

     $line =~ s/\,v\\//;
     $line =~ s/\,v//;
     chomp($line);
     print OUTF "$line \n";
}
close (INF);
close (OUTF);
```

Program 5.3

```
#!/usr/bin/perl

open (OUTF, ">>C:/perl/bin/mybugs4.txt") or die $!;

#This Program extracts all files with ".cp" or ".cpp"
#extention.

#!/usr/bin/perl

$count=0;

open (OUTF, ">>C:/perl/bin/finalbugs.txt") or die $!;

open (INF, "C:/perl/bin/mybugs4.txt")   or die $!;

while ($line= <INF>)

{

      if ($line =~ /\.cp/)

         {

           $count++;

           chomp($line);

           print OUTF "$line \n";

           print " $count $line \n ";

           }


 }

close (INF);

close (OUTF);
```

Program 5.4

```
#This program extracts all ".h" files .
#!/usr/bin/perl
$count=0;
open (OUTF, ">>C:/perl/bin/finalbugs.txt") or die $!;
open (INF, "C:/perl/bin/mybugs4.txt")   or die $!;
while ($line= <INF>)
     {
         if ($line =~ /\.h$/)
           {
             $count++;
             chomp($line);
             print " $count $line \n ";
             print OUTF "$line \n";
            }
      }
close (INF);
close (OUTF);
```

Program 5.5

```
# This program generates filenames along with their
# complete paths.

#!/usr/bin/perl

print "Which file do you want to edit?\n";

$filename = <STDIN>;

open (line, $filename) || die("Could not open file\n");

open (fpw, '>dir1.txt') || die("Could not open file\n");


$text = <line>;

$i=0;

while($text)
{
    if ($result=$text=~/.(\S+):/)
    {
     $path_name = $1;
    }
    elsif ($result=$text=~/^\w/)

    {
           chomp($text);

           print fpw ".$path_name:$text\n";

    }
    else

       {
         print fpw "\n";
       }

     $text=<line>;

}
```

Program 5.6

```
#!/usr/bin/perl

$count=0;

open (OUTF, ">>C:/perl/bin/finaldir.txt") or die $!;

open (INF, "C:/perl/bin/dir1.txt")   or die $!;

while ($line= <INF>)

   {

      if ($line =~ /\.h$/)

         {

             $count++;

             chomp($line);

             print " $count $line \n ";

             print OUTF "$line \n";

         }
}

close (INF);
close (OUTF);
```

Program 5.7

```
#This program extracts all ".cpp" filenames from dir1.txt
#!/usr/bin/perl
$count=0;
open (OUTF, ">>C:/perl/bin/finaldir.txt") or die $!;
open (INF, "C:/perl/bin/dir1.txt")   or die $!;
while ($line= <INF>)
      {
      if ($line =~ /\.cpp/)
           {
            $count++;
             chomp($line);
             print " $count $line \n ";
             print OUTF "$line \n";
           }
 }
close (INF);
close (OUTF);
```

Program 5.8

```
#This program reads the two text file named as finaldir.txt
#and finalbugs.txt and then writes the result of operation
#to the text file result.txt.

#!/usr/bin/perl

my $found;

open (OUTF, ">>C:/perl/bin/result.txt")   or die $!;

open (INF1,"C:/perl/bin/finaldir.txt")   or die $!;

open (INF2, "C:/perl/bin/finalbugs.txt")   or die $!;

my @data1;

my @data2;

@data1=<INF1>;

@data2=<INF2>;

foreach $data1(@data1)

{
      $found=0;

       foreach $data2(@data2)

          {
                  if ($data1=~m /^$data2$/)

                   {
                        $found++;
                   }
          }

      print OUTF " $data1 $found \n ";

}
print "\n";
```

Metrics-Delta Data set

Program-5.9

```
#This program reads the first text file generated from
#Mozmetrics report in, which data is given in the format
#of Filenames, Class names, Sloc and converts it as
#Filenames = Class names, Sloc. This conversion is done for
#the ease of creating data structure from a single text
#line


open (INF, "C:/perl/bin/ClassSloc.csv")   or die $!;

open (OUTF, ">>C:/perl/bin/classSloc1.txt")   or die $!;

while ($line= <INF>)

{
     @ary= split/,/,$line;

     print OUTF "$ary[0]=$ary[1],$ary[2]\n";

}
```

Program-5.10

```
#This Program is developed to read a text file in which
#data is in the format of Filename =Class name, Sloc and
#create a data structure named as Hash of Arrays and will
#convert data in the form of
#File name = Class name1, Sloc
#           Class name2, Sloc


open (INF, "C:/perl/bin/classSloc1.txt")   or die $!;

open (OUTF1, ">>C:/perl/bin/classSloc2.txt")   or die $!;

%hashofclasses=();

while ($line= <INF>)

  {
      ($class,$defect)=split/=/,$line;

        push (@{$hashofclasses{$class}},$defect);
  }

 for $class (keys %hashofclasses)

 {
     print OUTF1 "$class=\n @{ $hashofclasses{$class}}\n";
 }
```

Program-5.11

```
#This program is written to add all the Sloc in one one
#file named as tloc (total lines of program for all classes
#in a file) and divide Sloc of individual class with tloc.
#This division was performed to get percentage of a class
#in a file so that the defects are assigned according to
#their specific percentage. A class with higher value of
#Sloc will get higher portion of a defect. Greater the
#lines of code per class the greater the chances of errors
#are there. The output is
#File name, Class name, Contribution factor


open (INF, "c:/perl/bin/classSloc2.txt") or die $!;

open (OUTF, ">>c:/perl/bin/classSloc3.txt") or die $!;

%hashofclasses=();

while ($line=<INF>)

    {

       if ($line =~m /^\s\w/)

           {

               @ary=split/,/,$line;

               $aryclass[$n]=$ary[0];

                $n++;

                $aryclass[$n]=$ary[1];

                $n++;
           }


      $tloc=0;

      for $n(@aryclass)

           {
```

```
        if ($n=~m/^\d/)
          { $tloc=$tloc+$n;}
      }
if ($line =~m /^c:\/mozilla/)
      {   $filename=$line;
          @aryclass=();
      }
      if ($line =~m /^\n/)
     {
      print OUTF "$filename\n";
      for $n(@aryclass)
          {
            if ($n=~m/^\D/)
            {  print OUTF "$n\n";
            }
            if ($n=~m/^\d/)
            {
             $b=$n;
             $c=$b/$tloc;
             $c=sprintf("%.3f",$c);
             print OUTF "$c\n";
             }
          }
       }
```

Program-5.12

```
#This program reads each file name and its defects from
#delta data set, searches for that particular file from the
#results of Program-5.11 and when it finds out the file it
#is searching for, it multiplies the defect value to all
#the classes that belong to a file. Since the execution of
#program will assign the defects to individual classes
#there is no need to keep the data structure hash of array
#instead the out put will be in the format
#File name, Class name, Defects.

open (INF1,"C:/perl/bin/class-h-cpp.txt")   or die $!;

open (INF2, "C:/perl/bin/classSloc3.txt")   or die $!;

open (OUTF,">>C:/perl/bin/classSloc4.txt")   or die $!;

$count=0;

while ($line1= <INF1>)

  {
      $defects=0;

      if ($line =~m /^\d$/)

           {next;}

      $defects = readline (INF1);


     open (INF2, "C:/perl/bin/classSloc3.txt")   or die $!;

     while ($line2= <INF2>)

      {
          if ($line1=~m/^$line2$/)

            {
              $file=$line2;

               while ($line2= <INF2>)

                {

                   if ($line2=~m/^c:/)
```

```
        {last;}
      push @ary,$line2;
      foreach $_(@ary)
        {
          if ($_=~m/^\s\D/)
           {
            print OUTF "$file,$_";
           }
          if ($_=~m/^\d/)
           {
            $_=$_*$defects;
             print OUTF "$_\n";
           }
        }
    }
}
```

Program-5.13

```
#This program reads the first text file generated from
#Mozmetrics report in which the data is given in the format
#of Filenames, Class names :: Method name, Sloc and
#converts it as Filenames =Class names :: Method name,
#Sloc. This conversion is done for the ease of creating
#data structure from a single text line.
#File name =Class name : : Method name, Sloc


open (INF, "C:/perl/bin/methodSloc.csv")   or die $!;

open (OUTF, ">>C:/perl/bin/methodSloc1.txt")   or die $!;

while ($line= <INF>)

 {

       @ary= split/,/,$line;

       print OUTF "$ary[0]=$ary[1],$ary[2]\n";

 }
```

Program-5.14

```
#This program will convert the given data File name =  Class
#name:: Method name, Sloc  into hash of arrays. The file name
#is a hash key and the array contains class names followed by
#the method name and Sloc per method. The execution of this
#program extracts all the  classes (along with their methods
#and Sloc per method) that belong to #one file.
#File name =
#Class name1::Method name1, Sloc
#Class name2::Method name1, Sloc


open (INF, "C:/perl/bin/methodSloc1.txt")   or die $!;

open (OUTF1, ">>C:/perl/bin/methodSloc2.txt")   or die $!;

%hashofclasses=();

while ($line= <INF>)

  {

    ($class,$defect)=split/=/,$line;

     push (@{$hashofclasses{$class}},$defect); }

     for $class (keys %hashofclasses)

     {

    print OUTF1 "$class=\n @{ $hashofclasses{$class}

     }\n";

}
```

Program-5.15

```
#This Program extracts all the methods that belong to a
#particular class in a file and then it sums up Sloc for
#all the methods that belong to a single class. The result
#would be File name, Class name, Tsloc.
#Where Tsloc stands for total source lines of program for
#all methods that belong to a particular class in a single
#file


open (INF, "c:/perl/bin/methodSloc2.txt") or die $!;

open (OUTF, ">>c:/perl/bin/methodSloc3.txt") or die $!;

%hoh=();

while ($line= <INF>)

{

    if ($line =~m /^c:\//)

          {

             $filename=$line;

             %hashofclasses=();

             tloc=0;

             $ttloc=0;

             @defects=();

          }

     if ($line =~m /\:\:/)

          {

          ($class,$defect)=split/\:\:/,$line;
```

```
            push (@{$hashofclasses{$class}},$defect);
          }
        if ($line =~m /^\n/)
           {
            print OUTF "\n";
            print OUTF "$filename";
            for $class (keys %hashofclasses)
                 {
                    @defects= @{$hashofclasses{$class}
                 }
                    $tloc=0;
                    foreach $defects (@defects)
                     {
                          @loc=split/,/,$defects;
                          $nloc=$loc[1];
                          $tloc=$tloc+$nloc;

                     }
                     print OUTF "$class,$tloc\n";
                     $ttloc=$ttloc+$tloc;
              }
        }
}
```

Program-5.16

```
#This program adds Tsloc for all the classes in a file and
#divide each class by Ttsloc where Ttsloc stands for total
#source lines of Program for all classes in a file. This
#division is performed to get the percentage for assigning
#defects (we call it contribution factor)for each class in
#a file. File name, Class name, Contribution factor.


open (INF, "c:/perl/bin/methodSloc3.txt") or die $!;

open (OUTF, ">>c:/perl/bin/methodSloc4.txt") or die $!;

%hashofclasses=();

while ($line=<INF>)

 {

      if ($line =~m /^\s\w/)

              {

                   @ary=split/,/,$line;

                   $aryclass[$n]=$ary[0];

                   $n++;

                   $aryclass[$n]=$ary[1];

                   $n++;

              }

       if ($line =~m /^c:\/mozilla/)

              {

                  $filename=$line;
```

```
            %hashofclasses=();
            @aryclass=();
        }
if ($line =~m /^\n/)
        {
            print OUTF "$filename\n";
            $tloc=0;
            for $n(@aryclass)
                        {
                          if ($n=~m/^\d/)
                           { $tloc=$tloc+$n;}
                        }
             if ($tloc!=0)
                        {
                          for $n(@aryclass)
                            {
                             if ($n=~m/^\D/)
                             {print OUTF "$n\n";}
                             if ($n=~m/^\d/)
                             {$c=$n/$tloc;
                              $c=sprintf("%.2f",$c);
```

```
                print OUTF "$c\n";
            }
        }
    }
}
```

Program-5.17

```
#This program will assign defects count at the level of
#classes. For this assignment the delta data set is
#reopened, each file name and its respective defect count
#is read by the Program and assigned to the class that
#belong to this particular file by multiplying defects with
#contribution factor to obtain File name, Class name,
#Defects per Class

open (INF1,"C:/perl/bin/class-h-cpp.txt")   or die $!;

open (INF2, "C:/perl/bin/methodSloc4.txt")   or die $!;

open (OUTF,">>C:/perl/bin/methodSloc5.txt")   or die $!;

while ($line1= <INF1>)

  {

     $defects = 0;

     if ($line =~m /^\d$/)

     {next;}

     $defects = readline (INF1);

     open (INF2, "C:/perl/bin/method4.txt")   or die $!;

          while ($line2= <INF2>)

               {

                    if ($line1=~m/^$line2$/)

                     {

                      $file =$line2;

                      while ($line2= <INF2>)

                         {

                           if ($line2=~m/^c:/)
```

```
            {last;}
          push @ary,$line2;
        }
      foreach $_(@ary)
            {
               if ($_=~m/^\D/)
                {
                  print OUTF "$file,$_";
                }
                if ($_=~m/^\d/)
                {
                   $cd=$_*$defects;
                    print OUTF "$cd\n";

                }
            }
}
```

Program 5.18

```
#This program is developed to add defects count obtained
#from both classes and methods and get the resultant file.
#Input File: A.txt  [Figure 5.12]
#Input File: B.txt  [Figure 5.17]
#Output file: C.txt [Figure 5.19]


open (OUTF, ">>C:/perl/bin/classSloc7.txt")   or die $!;

open (INF1,"C:/perl/bin/classSloc6.txt")   or die $!;

open (INF2, "C:/perl/bin/methodSloc8.txt")   or die $!;

while ($data1= <INF1>)

{

if ($data1=~m/c:/)

 {

   @ary1=();

   @ary1=split/,/,$data1;

   $filename1=$ary1[0];

   $classname1=$ary1[1];

   $defects1=$ary1[2];

   open (INF2, "C:/perl/bin/methodSloc8.txt")   or die  $!;

   while ($data2= <INF2>)

          {

             if ($data2=~m/c:/)

                {

                  @ary2=();
```

```
                @ary2=split/,/,$data2;
                    $filename2=$ary2[0];
                    $classname2=$ary2[1];
                    $defects2=$ary2[2];
                }
              if ($filename1=~m/^$filename2$/
                 $classname1=~m/^$classname2$/)
                {
                 $defects1=$defects1+$defects2;
                 last;

                 }
          }
   chomp $filename1;chomp $classname1;chomp $defects1;
   print OUTF "$filename1,$classname1,$defects1\n";
   }
}
```

Program 5.20

```
#This program joins the defects count at the level of
#classes with their respective metric values

open (INF1,"C:/perl/bin/class-metrics.csv")   or die $!;

open (INF2, "C:/perl/bin/classSloc7.txt")   or die $!;

open (OUTF,">>C:/perl/bin/classSloc8.txt")   or die $!;

Label A : while ($data1= <INF1>)

          {

           if ($data1=~m/c:/)

           {

            @ary1=();

            @ary1=split/,/,$data1;

            $filename1=$ary1[0];

            $classname1=$ary1[1];

            $defects1=0;

            open(INF2,"C:/perl/bin/classSloc7.txt") or die
                 $!;

            Label A : while ($data2= <INF2>)

                      {

                        if ($data2=~m/c:/)

                          {
```

```
        @ary2=();
        @ary2=split/,/,$data2;
        $filename2=$ary2[0];
        $classname2=$ary2[1];
        $defects2=$ary2[2];
        }
if ($filename1=~m/^$filename2$/ &
   $classname1=~m/^$classname2$/)

{
  chomp
 $ary1[0],$ary1[0],$ary1[1],
 $ary1[2],$ary1[3],$ary1[4],
 $ary1[5],$ary1[6],$ary1[7],
 $ary1[8],$ary1[9],$ary1[10],
 $ary1[11],$ary1[12],$ary1[13],
 $ary1[14],$ary1[15],$ary1[16],
 $ary1[17],$ary1[18],$ary1[19],
 $ary1[20],$ary1[21],$ary1[22]
 ,$ary1[23],$defects2;
 print OUTF
"$ary1[0],$ary1[1],$ary1[2],
 $ary1[3],$ary1[4],$ary1[5],
 $ary1[6],$ary1[7],$ary1[8],
```

```
                $ary1[9],$ary1[10],$ary1[11],
            $ary1[12],$ary1[13],$ary1[14],
            $ary1[15],$ary1[16],$ary1[17],
            $ary1[18],$ary1[19],$ary1[20],
            $ary1[21],$ary1[22],$ary1[23],
            $defects2";
            last;

        }

    }

}
```